TECHNICAL REPORT

R. S. Fabry and C. Sequin

"The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, or the Defense Advanced Research Projects Agency of the U.S. Government."

Contract N00039-82-C-0235
November 15, 1981 - September 30, 1983

ARPA Order Number 4031

PERFORMANCE ANALYSIS OF DISTRIBUTED DATA BASE SYSTEMS

by

Michael Stonebraker, John Woodfill, Jeff Ranstrom,
Joseph Kalash, Kenneth Arnold and Erika Andersen

DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE
UNIVERSITY OF CALIFORNIA
BERKELEY, CA.

ABSTRACT

In this paper we briefly present the design of a distributed relational data base system. Then, we discuss experimental observations of the performance of that system executing both short and long commands. Conclusions are also drawn concerning metrics that distributed query processing heuristics should attempt to minimize. Lastly, we comment on architectures which appear viable for distributed data base applications.

• Research Sponsored by the National Science Foundation under Grant MCS-8211528 and by the Defense Advanced Research Projects Agency under Contract N00039-C-0235.

1. INTRODUCTION

Many algorithms have been proposed to solve distributed relational data base problems in the areas of:

a) distributed concurrency control

b) distributed crash recovery

c) support of multiple copies of data

d) distributed command processing

There is currently little quantitative knowledge on the performance of such algorithms. Previous work has been based exclusively on simulation, *e.g.* [RIES79, GARC79a, GARC79b, LIN81] or formal modeling, *e.g.* [GELE78, BERN79]. One of the objectives of this research is to provide empirical results concerning the performance of various algorithms.

This paper first presents a short description of a working prototype distributed data base system. Then, we present the results of a collection of experiments on this prototype. Conclusions concerning query processing algorithms are drawn as appropriate. Lastly, comments on viable distributed architectures for data base applications are presented.

2. DISTRIBUTED INGRES

Distributed INGRES operates on a collection of DEC VAX 11/780s and 11/750s connected by a 3 mbit ethernet. All run the 4.1cBSD, a version of the UNIX [RITC75] operating system enhanced at Berkeley with paging, numerous program development tools, remote interprocess communication, and remote process execution.

Most features of Distributed INGRES [EPST78] are currently operational. A master INGRES process runs at the site where the command originated and slave INGRES processes run at each site which have data involved in the command. The

master process parses the command, resolves any views, and creates an action plan to solve the command using the fragment and replicate technique. The slave process is essentially single-machine INGRES [STON76, STON80] with minor extensions and with the parser removed. The coordinator and slaves communicate using the 4.1cBSD interprocess message system.

Distributed INGRES supports fragments of relations at different sites. For example, one can distribute the relation

EMP (name, salary, manager, age, dept)

as follows:

```
range of E is EMP
distribute E
  at Berkeley where E.dept = "shoe"
  at Paris    where E.dept = "toy"
  at Boston   where E.dept != "toy" and
                E.dept != "shoe"
```

Berkeley, Paris and Boston are logical names of machines which are mapped to site addresses by a lookup table. A single site relation is a special case of the distribute command, *e.g.*

```
distribute ONE-SITE at Berkeley
```

Currently, all QUEL commands without aggregates are processed correctly for distributed data. Consider, for example, the following update:

```
range of E is EMP
replace E(dept = "toy") where e.salary > 10000
```

This command will be processed by all sites containing fragments of the EMP relation. All qualifying tuples are updated and their site location may be changed. For example, the tuple of an employee earning more than 10000 in the shoe department would be moved from Berkeley to Paris.

Distributed INGRES uses a two phase commit protocol [GRAY78, LAMP76]. Slaves send a "ready" message to the master when they are prepared to commit

an update. Tuples which change sites are included with this message. The master then redistributes the tuples by piggybacking them onto the commit message. A three phase commit protocol can optionally be used [SKEE82] for added reliability. In this case the above redistribution is handled in the second phase.

When a command spans data at multiple sites, a rudimentary version of the "fragment and replicate" query processing strategy is used. For example, suppose a second relation

```
DEPT (dname, floor, budget)
```

exists at two sites as follows:

```
distribute D
 at Berkeley where D.budget > 5
 at Paris    where D.budget <= 5
```

Consider the query submitted by a Boston user:

```
range of E is EMP
range of D is DEPT
retrieve (E.name) where E.dept = D.dname
                  and D.floor = 1
```

First, the one variable clause "D.floor = 1" is detached and run at both Berkeley and Paris, i.e.

```
range of D is DEPT
retrieve into TEMP (D.dname) where D.floor = 1
```

The original query now becomes

```
range of E is EMP
range of D is TEMP
retrieve (E.name) where E.dept = D.dname
```

To satisfy the query, data movement must now take place. One relation (say TEMP) is replicated at each processing site. Hence, both Berkeley and Paris send their TEMP relations to each site which has a fragment of EMP. Therefore, the needed transmissions are:

```
TEMP(Paris) -> Boston
```

TEMP(Paris) -> Berkeley
TEMP(Berkeley) -> Paris
TEMP(Berkeley) -> Boston

Now, all three sites have a complete copy of TEMP and a fragment of the EMP relation. The above query is now performed at each site, and the resulting tuples are returned to the master site, where they are displayed to the user.

Since our ETHERNET has the hardware capability to support broadcast, it is possible to perform the above four transfers by broadcasting each fragment of TEMP. However, the 4.1cBSD operating system does not support multicast or broadcast transmissions. Consequently, the above four transmissions occur separately, and the strategy of replication may perform poorly [EPST78]. The network on which we planned to run [ROWE79] supported broadcast, and we have not subsequently modified the query processing heuristics.

At the moment, the relation to be replicated is chosen arbitrarily, so TEMP and EMP are equally likely to be selected for movement. A more elegant strategy is being planned.

3. SIMPLE UPDATES

In all experiments we use the EMP and DEPT relations as discussed in Section 1. Our data base consists of 30,000 EMP tuples, each 36 bytes long and 1500 DEPT tuples each 18 bytes long. In all cases we will be comparing the performance of Distributed and single-site INGRES.

The first benchmark consists of 1000 random updates of the form:

replace E (salary = K) WHERE E.name = L

The n-site data base was distributed as follows:

distribute E
at *site-a* where e.name < j_1
at *site-b* where e.name >= j_1
and e.name < j_2
.
.

at site-n where e.name >= f_{n-1}

The constants $f_1, \ldots f_{n-1}$ were chosen so that exactly $1000/n$ updates were directed to each site. The number of sites, n, was varied from 1 to 3.

Each site ran a script which contained $1000/n$ updates and processed the next command when it received "done" from the previous one. In this way there was a master INGRES at each site and we avoided creating a bottleneck at a single coordinating site.

Note that this benchmark consists of a large collection of small transactions, each of which can be completely processed at a single site. A distributed data base should perform well in this situation.

Table 1 indicates for each configuration the CPU time spent inside the operating system, the CPU time spent inside the INGRES code and the elapsed time. The benchmark was run on a VAX 11/780 along with 0, 1 or 2 VAX 11/750's. Unfortunately, the 11/750's have varying amounts of main memory, disk systems, and buffer space allocations. Moreover, the error rate of network transmission varies between pairs of machines. As a result, a fair amount of random variation of the numbers must be expected.

For the distributed processing configurations, the reported times are a sum of the time spent by the master INGRES at that site along with the times spent by any slave INGRESs on behalf of masters at other sites. According to local benchmarks, an 11/750 is about 0.629 times as fast as an 11/780 [HAGG83]; hence total CPU time is calculated by scaling 11/750 time by the above factor and is reported in the row labeled by n*780.

Several conclusions can be drawn from these results. First, Distributed INGRES is about 20 percent slower than normal INGRES when run on a local data base. Distributed INGRES must check the distribution criteria to ascertain that

	user time	system time	elapsed time
Normal INGRES			
11/780	7:34	3:04	22:34
Distributed INGRES - local data base			
11/780	9:06	3:53	26:57
Distributed INGRES - foreign data base			
11/750	7:58	3:02	28:37
11/780	5:34	2:57	
02*780	10:35	4:51	
Distributed INGRES - 2 sites			
11/780	5:14	2:24	15:30
11/750	8:24	4:05	16:46
02*780	10:31	4:58	
Distributed INGRES - three sites			
11/780	3:43	1:30	12:43
11/750	5:28	2:16	13:34
11/750	5:48	2:13	13:22
03*780	11:09	4:30	

Performance of Simple Updates
Table 1

each of the commands is a local one. Currently, this checking is performed at *run* time; however, for better performance it could be performed at compilation time. In addition, each updated tuple must also be checked against the distribution criteria to ensure that it does not change sites (i.e. that the dept field is not being changed).

Second, Distributed INGRES on a one machine foreign data base is about 10 percent slower than on a local data base. The foreign data requires master INGRES to communicate with a non-local slave instead of a local slave, and this requires extra user and system CPU time.

Third, 2*780 and 3*780 Distributed INGRES use 20 percent more CPU time than Distributed INGRES on a local data base and 45 percent more CPU time than single-site INGRES. Both systems use marginally more CPU time than Distributed INGRES on a one-site foreign data base. The benefit of these configurations is increased parallel processing; hence the benchmark finishes respectively 25 and 40 percent faster. Of course, the benchmark would have finished even faster if the additional machines were 11/780s. We suspect that a collection of n 11/780s could finish the benchmark in approximately 28/n minutes.

Lastly, note that the 3 site benchmark uses the same amount of CPU time as the two site benchmark. It is reasonable to expect that the total CPU time would continue to be a constant as additional sites were added. Hence, we predict that total aggregate CPU time would remain a constant as sites are added and would be split among an increasing number of machines.

Benchmark 1 on a foreign data base results in 522,880 bytes being transferred across the network, and less than two percent of the available bandwidth is consumed. It appears that a large number of machines could be added to the ETHERNET before bandwidth limitations arise.

4. ONE RELATION RETRIEVES

In this benchmark we attempted to load the network as fully as possible with the following query:

```
range of E is EMP
retrieve (E.all)
```

The result of this query is 30000 tuples which would ordinarily be printed on the terminal. To stress differences in the environments being tested, we discarded the qualifying tuples in both this benchmark and the subsequent one. Hence, the cost of printing more than 1 mbyte of data is not included in the results

presented in Table 2. In the 2 site and 3 site benchmarks the EMP relation is uniformly distributed across the sites. Moreover, we are timing several repetitions of the query submitted from a single job stream and then averaging them.

Distributed INGRES on a local data base runs at about the same speed as single-site INGRES. The extra overhead of discovering that the query is local is amortized over a large amount of processing, so the two systems perform comparably.

	user time	system time	elapsed time
Normal INGRES			
11/780	1:44	0:15	2:03
Distributed INGRES - local data base			
11/780	1:47	0:10	2:05
Distributed INGRES - foreign data base			
11/750	0:03	0:03	2:54
11/780	1:47	0:20	
02*780	1:49	0:22	
Distributed INGRES - 2 sites			
11/780	1:06	0:19	
11/750	1:36	0:15	2:59
02*780	2:06	0:28	
Distributed INGRES - three sites			
11/780	0:35	0:05	2:37
11/750	1:13	0:16	
11/750	1:12	0:17	
03*780	2:06	0:26	

Performance of One-relation Retrieves
Table 3

On a foreign data base distributed INGRES is 0:49 seconds slower. In this environment, a slave must write the EMP relation into a temporary file and pass it across the network to a another file. Consequently, there are a total of two copies made of the 1200 block EMP relation.

The cost of a executing a remote copy of the 1200 block file is 0:17 of elapsed time and 0:11 of system CPU time. Hence, about 32 percent of the 0:49 difference is explained by the network overhead; the rest is added INGRES overhead. This remote copy consumes about 19.3 percent of the 3 mbit bandwidth. Because INGRES adds extra overhead, it uses only 6 percent of the available bandwidth. Obviously a large number of concurrent data base users would be required before INGRES could use any substantial fraction of the ETHERNET bandwidth.

When the data base is distributed over multiple sites, the total CPU time remains approximately constant and is distributed evenly over the machines. When two sites are present, about 50 percent of the CPU cycles are offloaded to an 11/750 which is 0.629 times as fast. The maximum improvement possible in this configuration is about 25 percent, and it appears that INGRES overhead offsets this gain. With three sites dividing the work, response time begins to improve, and this improvement should continue as new sites are added.

Four conclusions can be drawn from the results of this benchmark and the above discussion. First, query processing heuristics should account for the speed of the various machines when deciding optimal strategies. To achieve minimum response time using our configuration, one should give the 11/780 disproportionately more work than the 11/750s. Second, bandwidth will never be a problem in our environment. Even operating system file transfers do not come close to using the entire bandwidth, and INGRES relations cannot be moved any faster than OS files. Third, data base and file servers are often pro-

posed as useful architectural concepts in a local network environment. However, this configuration is closely approximated by a foreign data base which had the next to worst performance of the ones tested. Unless a server is much faster than other machines on the network or unless other machines do not have disks, the merits of a server seem doubtful. Lastly, it appears desirable to split complex queries among a large number of sites and take advantage of the resulting parallel processing.

5. JOIN EXPERIMENT

The last experiment executed the natural join of EMP and DEPT, with EMP hashed on the dept field and DEPT hashed on the dname field, i.e:

```
range of E is EMP
range of D is DEPT
retrieve (E.all, D.all) where E.dept = D.dname
```

The same environments were tested as in the previous sections. In the 2 and 3 site cases both EMP and DEPT were uniformly distributed, and DEPT was selected as the relation to be replicated in query processing. Table 4 contains the measured results.

Notice that these results are very similar to the preceding two sets of numbers. Hence, we will not comment on their relative magnitudes. Rather, we will discuss other points.

First, the two and three site versions moved the DEPT relation to solve the query. We forced distributed INGRES to instead move the EMP relation, and the results were about 20 times slower than those reported. The explanation is somewhat subtle. When Distributed INGRES replicates a relation at multiple sites, it loses the access structure of the relation involved and does not recreate the original access path for the composite relation. Hence, if DEPT or EMP is moved, it becomes a heap at each site. Local INGRES algorithms solve the join by iterating over the smaller of the two relations, in this case DEPT. If DEPT is

	user time	system time	elapsed time
Normal INGRES 11/780	8:41	0:38	9:37
Distributed INGRES - local data base 11/780	8:57	0:47	10:34
Distributed INGRES - foreign data base 11/750			
11/780	9:01	0:42	
02*780			
Distributed INGRES - 2 sites 11/780	4:28	:21	10:45
11/750	7:56	1:02	
02*780	9:28	1:00	
Distributed INGRES - three sites 11/780	3:11	:13	7:41
11/750	5:27	:43	
11/750	5:14	:40	
03*780	9:54	1:05	

Performance of Joins
Table 4

moved, then INGRES will iterate over a heap producing a large collection of queries of the form:

```
retrieve (E.all, -constants- )
  where E.dept = constant
```

These queries can then be executed by a hashed access to the EMP relation. However, if EMP is moved and becomes a heap, a large number of queries are generated, each requiring a complete scan of the EMP relation.

We did not execute the query with EMP at one site and DEPT at another. In this case the query processing module should move the DEPT relation to the site of EMP. This should add only a few seconds of overhead to the distributed

INGRES times for a local data base.

We also did not force the obvious semi-join strategy indicated by the following commands.

```
retrieve into W(E.dept)
move W
retrieve into W2 (D.all) where D.dname = W.dept
move W2
retrieve (E.all, W2.all) where E.dept = W2.dname
```

Since all values of dname appear in the EMP relation, W2 is exactly the size of DEPT. This algorithm will consequently perform more poorly than all other ones since it will perform a projection of the EMP relation in addition to the work done by other algorithms. Given that bandwidth is not a consideration in our environment, semi-joins, which must execute the query twice, will seldom be advantageous.

6. CONCLUSIONS

This paper presented timings for a distributed data base system. By and large, they are extremely encouraging. Although Distributed INGRES is not highly optimized, it does not add a large amount of overhead. It is expected that judicious tuning could make it competitive with single-site INGRES on local data bases. On distributed data, the costs of moving data are not excessive and result in substantial parallelism.

REFERENCES

[BERN79] Bernstein, P. and Chiu, D., "Using Semi-joins to Solve Relational Queries", Computer Corp. of America, Cambridge, Mass., Jan. 1979.

[EPST78] Epstein, R., et. al., "Distributed Query Processing in a Relational Data Base System," Proc. 1978 ACM-SIGMOD Conference on Management of Data, Austin, Texas, May, 1978.

[GARC79a] Garcia-Molina, H., "Centralized Control Update Algorithms for Fully Redundant Distributed Data Bases," Proc. 1st International Conference on Distributed Computing, Huntsville, Ala., Oct. 1979.

[GARC79b] Garcia-Molina, H., "Performance of Update Algorithms for Replicated Data in a Distributed Data Base," PhD Thesis, Stanford University, Computer Science Dept, June 1979.

[GELE78] Gelenbe, E. and Sevcik, K., "Analysis of Update Synchronization for Multiple Copy Data Bases," Proc. 3rd Berkeley Workshop on Distributed Data Bases and Computer Networks, San Francisco, Ca., February 1978.

[GRAY78] Gray, J., "Notes on Data Base Operating Systems," in Operating Systems: An Advanced Course, Springer-Verlag, 1978, pp393-481.

[HAGG83] Hagmann, R., private communication

[LAMP76] Lampson, B. and Sturgis, H., "Crash Recovery in a Distributed System," Xerox Palo Alto Research Center, 1976.

[LIN81] Lin, W., "Performance Evaluation of Two Concurrency Control Mechanisms in a Distributed Data Base System," Proc. 1981 ACM-SIGMOD Conference on Management of Data, Ann Arbor, Mich., May 1981.

[RIES79] Ries, D., "The Effects of Concurrency Control on Data Base Management System Performance," Electronics Research Laboratory, Univ. of California, Memo ERL M79/20, April 1979.

[RITC75] Ritchie, D. and Thompson, K., "The UNIX Timesharing System," CACM, June 1975.

[ROWE79] Rowe, L. and Birman, K., "Network Support for a Distributed Data Base System", Proceedings of the Fourth Berkeley Workshop on Distributed Data Management and Computer Networks, August, 1979, San Francisco, California.

[SKEE82] Skeen, D., "A Quorum-Based Commit Protocol," Proc. 6th Berkeley Workshop on Distributed Data Bases and Computer Networks, Pacific Grove, Ca., Feb 1982.

[STON76] Stonebraker, M. et. al., "The Design and Implementation of INGRES," TODS 2, 3, September 1976.

[STON80] Stonebraker, M., "Retrospection on a Data Base System," TODS, March 1980. eqn fatal error: can't open file tbl file tbl, between lines 716 and 1

IMPLEMENTATION OF RULES IN RELATIONAL DATA BASE SYSTEMS

by
Michael Stonebraker, John Woodfill and Erika Andersen

Dept of Electrical Engineering and Computer Science
University of California
Berkeley, Ca.

ABSTRACT

This paper contains a proposed implementation of a rules system in a relational data base system. Such a rules system can provide data base services including integrity control, protection, alerters, triggers, and view processing. Moreoever, it can be used for user specified rules. The proposed implementation makes efficient use of an abstract data type facility by introducing new data types which assist with rule specification and enforcement.

I INTRODUCTION

Rules systems have been used extensively in Artificial Intelligence applications and are a central theme in most expert systems such as Mycin [SHOR76] and Prospector [DUDA78]. In this environment knowledge is represented as rules, typically in a first order logic representation. Hence, the data base for an expert system consists of a collection of logic formulas. The role of the data manager is to discover what rules are applicable at a given time and then to apply them. Stated differently, the data manager is largely an inference engine.

On the other hand, data base management systems have tended to represent all knowledge as pure data. The data manager is largely a collection of heuristic search procedures for finding qualifying data. Representation of first order logic statements and inference on data in the data base are rarely attempted in production data base management systems.

The purpose of this paper is to make a modest step in the direction of supporting logic statements in a data base management system. One could make this step by simply adding an inference engine to a general purpose DBMS. However, this would entail a large amount of code with no practical interaction with the current search code of a data base system. As a result, the DBMS would get much larger and would contain two essentially non overlapping subsystems. On the other hand, we strive for an implementation which integrates rules into DBMS facilities so that current search logic can be employed to control the activation of rules.

The rules system that we plan to implement is a variant of the proposal in [STON82], which was capable of expressing integrity constraints, views and protection as well as simple triggers and alarms for the relational DBMS INGRES [STON76]. Rules are of the form:

on condition
then action

The conditions which were specified include:

the type of command being executed (e.g. replace, append)

the relation affected (e.g. employee, dept)
the user issuing the command
the time of day
the day of week
the fields being updated (e.g. salary)
the fields specified in the qualification
the qualification present in the user command

The actions which we proposed included:

sending a message to a user
aborting the command
executing the command
modifying the command by adding qualification or
changing the relation names or field names

Unfortunately, these conditions and actions often affect the command which the user submitted. As such, they appear to require code that manipulates the syntax and semantics of relational commands. This string processing code appears to be complex and has little function in common with other data base facilities. In this paper we make use of two novel constructs which make implementing rules a modest undertaking. These are:

1) the notion of executing the data
and
2) a sequence of QUEL commands as a data type for a relational data base system

The remainder of this paper is organized as follows. In Section II we indicate the new data types which must be implemented and the operations required for them. Then in Section III we discuss the structural extensions to a relational data base system that will support rules execution. Lastly, Section IV and V contains some examples and our conclusions.

II RULES AS ABSTRACT DATA TYPES

Using current INGRES facilities [FOGG82, ONG82, STON82a] new data types for columns of a relation can be defined and operators on these new types specified. We use this facility to define several new types of columns and their associated operators in this section.

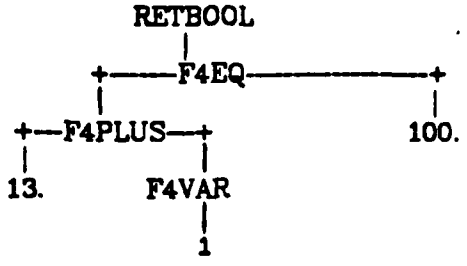
The first data type is a QUEL command, e.g.

```
range of e is employee
replace e(salary = 1.1*e.salary) where e.name = "John"
```

The abstract data type facility supports an external representation such as that above for a given data type. Moreover, when an object of the given type is stored in the data base it is converted to an internal representation. QUEL commands are converted by the INGRES parser to a parse tree representation such as the one noted in Figure 1 for the qualification "where 13. + employee.salary = 100". Consequently, a natural internal form for an object of type QUEL is a parse tree. Each node in this parse tree contains a value (e.g. 13.) and a type (e.g. floating point constant).

The second new data type which will be useful is an ATTRIBUTE-FUNCTION. This is a notion in the QUEL grammar and stands for anything that can be evaluated to a constant or the name of a column. Examples of attribute functions include:

13.



The Parse Tree for the Qualification
Where 13. + employee.salary = 100

Figure 1

1.1*employee.salary +20

newsal

The external representation is the same string format used for objects of type QUEL; the internal representation is that of a parse tree.

Two other data types of lesser significance are also needed, a TIME data type to contain a time of day value and a COMMAND data type to contain a value which is one of the QUEL commands.

Current built-in INGRES operators (e.g. *, /, +, etc.) must be extended for use with attribute functions. In addition, two new operators are also required. First, we need a function new() which will operate with integer data types. When called, it will return a new unique identifier which has not been previously used. Second, we require a partial match operator, ~, which will operate on a variety of data types and provide either equality match or match the value "*".

III INGRES CHANGES

We expect to create two rules relation, RULES1 and RULES2, with the following fields:

```
create RULES1(
    rule-id = i4,
    user-id = c10,
    time    = time,
    command = command,
    relation = c12,
    terminal = c2,
    action = quel)

create RULES2 (
    rule-id = i4,
    type = c10,
    att-fn1 = attribute-function
    operator = c5,
    att-fn2 = attribute-function)
```

For example, we might wish a rule that would add a record to an audit trail

whenever the user "Mike" updated the employee relation. This requires a row in RULES1 specified as follows:

```
append to RULES1(
    rule-id  = new(),
    user-id  = "Mike",
    command  = "replace",
    relation = "employee",
    action   = QUEL command to perform audit)
```

If additionally we wished to perform the audit action only when Mike updated the employee relation with a command containing the clause "where employee.name = "Fred"" we would add an additional tuple to RULES2 as follows:

```
append to RULES2(
    rule-id  = the one assigned in RULES1
    type     = "where"
    att-fn1  = "employee.name"
    operator = "="
    att.fn2  = "Fred")
```

We also require the possibility of executing data in the data base. We propose the following syntax:

```
range of r is relation
execute (r.field) where r.qualification
```

In this case the value of r.field must be an executable QUEL command and thereby of data type QUEL. To execute the rule that was just appended to R1 we could type:

```
range of r is R1
execute (r.action) where r.user-id = "Mike" and
                         r.command = "replace" and
                         r.relation = "employee"
```

When a QUEL command is entered by a user, it is parsed into an internal parse tree format and stored in a temporary data structure. We expect to change that data structure to be the following two main memory relations:

```
create QUERY1(
    user-id = c10,
    command = command,
    relation = c12,
    time    = time,
    terminal= c2)

create QUERY2(
    clause-id = i4,
    type      = c10,
    att-fn1   = attribute-function,
    operator  = c5,
    att-fn2   = attribute-function)
```

If the user types the query:

```
range of e is employee
retrieve (e.salary)
      where (e.name = "Mike" or e.name = "Sally")
        and e.salary > 30000
```

then INGRES will build QUERY1 to contain a single tuple with values:

QUERY1				
user-id	command	relation	time	terminal
current-user	retrieve	employee	current-time	current-terminal

QUERY2 will have four tuples as follows:

QUERY2				
clause-id	type	att-fn1	operator	att-fn2
id-x	target-list	employee.salary	=	employee.salary
id-y	where	employee.name	=	Mike
id-y	where	employee.name	=	Sally
id-z	where	employee.salary	>	30000

Notice that QUERY1 and QUERY2 contain a relational representation of the parse tree corresponding to the incoming query from the user. The where clause of the query is stored in conjunctive normal form, so that atomic formulae which are part of a disjunction have the same clause-id, while the atomic formulae and disjunctions in the conjunction have different clause-ids.

Then we execute the QUEL commands in Figure 2 to identify and execute the rules which are appropriate to the incoming command. These commands are performed by the normal INGRES search logic. Activating the rules system simply means running these commands prior to executing the user submitted command. After running the commands of Figure 2, the query is converted back to a parse tree representation and executed. Notice that the action part of a rule can update QUERY1 and QUERY2; hence modification of the user command is easily accomplished. The examples in the next section illustrate several uses for this feature:

```
range of r1 is RULES1
range of r2 is RULES2
range of q1 is QUERY1
range of q2 is QUERY2
retrieve into TEMP(r1.id, r1.quel) where
        r1.user-id ~ q1.user-id and
        r1.command ~ q1.command and
        r1.time    ~ q1.time    and
        r1.terminal~ q1.terminal

range of t is TEMP
execute (t.quel) where t.id < 0 or
       (t.id = r2.rule-id and
        set(r2.all-but-rule-id by r2.rule-id)
      = set(r2.all-but-clause-id by r2.rule-id
        where r2.all-but-rule-id ~ q2.all-but-clause-id))
```

Rule Activation in QUEL
Figure 2.

The set functions are as defined in [HELD75]. The conditions for activating a rule are:

(i) its tuple in RULES1 matches the tuple in QUERY1

and either

(ii) each tuple for the rule in RULES2 matches a tuple in QUERY2.

or

(iii) there are no required matches in RULES2
(represented by rule-id < 0).

The second condition provides appropriate rule activation when both the user query and the rule do not contain the boolean operator OR. However, a rule which should be activated when two clauses A and B are true will have two tuples in RULES2. This rule will be activated by a user query containing clauses matching A and B connected by any boolean operator. Under study is a more sophisticated activation system which will avoid this drawback.

The commands in Figure 2 cannot be executed directly because set functions have never been implemented in INGRES. Hence, we turn now to a proposed implementation of these functions.

Suppose we define a new operator "|" to be bitwise OR, and "bitor()" to be an aggregate function which bitwise ORs all qualifying fields. Then if we add the attribute "mask" to RULES2, and give each tuple for a particular rule a unique bit, the following query is correct:

```
range of t is TEMP
execute (t.quel) where t.id < 0 or
        (t.id = r2.rule-id and
      bitor(r2.mask by r2.rule-id)
      = bitor(r2.mask by r2.rule-id
         where r2.all-but-rule-id ^ q2.all-but-clause-id))
```

This solution will be quite slow, since the test for each rule involves processing a complicated aggregate. A more efficient solution involves generating masks for all rules in parallel and writing special search code as follows:

```
range of r1 is RULES1
range of r2 is RULES2
range of q1 is QUERY1
range of q2 is QUERY2
retrieve into TEMP(r1.id, r1.quel, mask = 0)  where
          r1.user-id ^ q1.user-id and
          r1.command ^ q1.command and
          r1.time    ^ q1.time    and
          r1.terminal^ q1.terminal

range of t is TEMP

foreach q2 do begin
   replace t (mask = t.mask | r2.mask)
            where t.id = r2.rule-id and
            r2.all-but-rule-id ^ q2.all-but-clause-id
end foreach

execute (t.quel) where t.id < 0 or
                (t.id = r2.rule-id and
             bitor(r2.mask by r2.rule-id)
                = t.mask)
```

Since the value of "bitor(r2.mask by r2.ruleid)" remains constant, the performance of this algorithm can be further improved by including the value of "bitor(r2.mask by r2.ruleid)" in RULES1 and copying it into TEMP as the "acceptmask". The third query would then become:

```
execute (t.quel) where t.id = r2.rule-id and
        t.acceptmask = t.mask
```

Notice the case where there are no tuples in RULES2 for a particular rule is handled with an acceptmask of zero.

Either a variable length abstract data type "bitstring" or a four byte integer can be used to store the mask. The abstract data type solution has the advantage of allowing an unlimited number of conditions for specifying rule activation, while the four byte integer solution has the advantage of simplicity and speed, but can only represent 32 conditions.

IV EXAMPLES

We give a few examples of the utility of the above constructs in this section. First, we can store a command in the data base as follows:

```
append to storedqueries (id = 6,
        quel = "range of e is employee
             retrieve (e.salary)
             where e.name = "John"")
```

We can execute the stored command by:

```
range of s is storedqueries
execute (s.quel) where s.id = 6
```

The following two examples will pertain to the query:

```
range of e is employee
replace e(salary = salary*1.5) where e.name = "Erika"
```

To represent this query INGRES will append the following tuples to the QUERY1 and QUERY2 relations:

QUERY1				
user-id	command	relation	time	terminal
current-user	replace	employee	current-time	current-terminal

QUERY2				
clause-id	type	att-fn1	operator	att-fn2
id-z	target-list	employee.salary	=	employee.salary*1.5
id-x	where	employee.name	=	Erika

Suppose we want to implement the integrity contraint to insure that employee salaries never exceed $30,000. Using query modification [STON75] we would add the clause "and employee.salary*1.5 <= 30000". to the user's qualification with the following rule:

```
append to RULES1(
    rule-id  =  new(),  (call it id-y)
    user-id  =  *,      (matches any user-id)
    command  =  "replace",
    relation =  "employee",
    action   =  "range of Q2 is QUERY2
                append to QUERY2(
                 clause-id = id-x,
                      type     = "where",
                      att-fn1  = Q2.att-fn2,
                      operator = "<=",
```

```
                    att-fn2  = "30000")
                where Q2.att-fn1 = "employee.salary")"
append to RULES2(
    rule-id  = id-y,
    type     = "target-list",
    att-fn1  = "employee.salary",
    operator = "=",
    att-fn2  = *)
```

Consider a transition integrity constraint that specifies that the maximum salary increase is 20%. This means that the new salary divided by the old salary must be less than or equal to 1.2. This can be achieved by appending a single tuple to R1:

```
append to RULES1(
    rule-id  = new(),
    user-id  = *,
    command  = "replace",
    relation = "employee",
    action   = "range of Q2 is QUERY2
               append to QUERY2(
                clause-id = id-x,
                    type     = "where",
                    att-fn1  = Q2.att-fn2/Q2.att-fn1,
                    operator = "<="
                    att-fn2  = "1.2")
           where Q2.att-fn1 = "employee.salary""
```

As a last example of an integrity constraint, consider a referential constraint that a new employee must be assigned to an existing department. Such a rule would be applied, for example, to the following query:

```
append to employee (name="Chris", dept = "Toy", mgr = "Ellen")
```

The corresponding tuples in QUERY2 would look like:

QUERY2				
clause-id	type	att-fn1	operator	att-fn2
id-z	target-list	employee.name	=	Chris
id-z	target-list	employee.dept	=	Toy
id-z	target-list	employee.mgr	=	Ellen

Implementation of the constraint requires checking that the department given in the target list of the append appears in the department relation. This is accomplished with the following rule:

```
append to RULES1(
    rule-id  = new(),
    user-id  = *,
    command  = "append",
    relation = "employee",
    action   = "range of Q2 is QUERY2
               append to QUERY2(
                    clause-id  = id-z,
                    type       = "where",
                    att-fn1    = "dept.name",
                    operator   = "=",
                    att-fn2    = Q2.att-fn2)
```

where Q2.att-fn1 = employee.dept"

Lastly, protection is achieved primarily by making use of the RULE1 relation, which pertains to the query "bookkeeping" information. Suppose we wanted to ensure that no one could access the employee relation after- hours (after 5PM and before 8AM). The following tuple would be added to the R1 relation:

```
append to RULES1(
    rule-id  = new(),
    user-id  = *,
    time     = "17:01 - 7:59",
    command  = *,
    relation = "employee",
    terminal = *,
    action   = "range of Q1 is QUERY1
               range of Q2 is QUERY2
               delete Q1
               delete Q2
```

If the query meets the conditions, the action removes the tuples in QUERY1 and QUERY2 and thereby aborts the command.

V CONCLUSIONS

This paper has presented an initial sketch of a rules system that can be embedded in a Relational DBMS. There are two potentially very powerful features to our proposal. First, it can provide a comprehensive trigger and alerter system. Real time data base applications, especially those associated with sensor data acquisition, need such a facility. Second, it provides stored DBMS commands and the possibility of parallel execution of triggered actions. In a multiprocessor environment such parallelism can be exploited.

There are also several deficiencies to the current proposal, including:

a) Rule specfication is extremely complex. This could be avoided by a language processor which accepted a friendlier syntax and translated it into the one in this paper.

b) The result of the execution of a collection of rules can depend on the order in which they are activated. This is unsettling in a relational environment.

c) Rules trigger on syntax alone. For example, if we want a rule that becomes activated whenever John's employee record is affected, we trigger on any query having "employee.name = John" in the where clause. However if the incoming query is to update all employees' salaries, this rule would not be triggered.

d) Commands with multiple range variables over the same relation, so called *reflexive joins*, are not correctly processed by the rules engine.

e) Aggregate functions have not yet been considered.

f) As noted earlier, boolean OR is not treated correctly.

We are attempting to resolve these difficulties with further work.

REFERENCES

[DUDA78] Duda, R. et. al., "Development of the Prospector Consultation System for Mineral Exploration," SRI International, October 1978.

[FOGG82] Fogg, D., "Implementation of Domain Abstraction in the Relational Database System, INGRES", Masters Report, EECS Dept, University of California, Berkeley, CA Sept. 1982.

[HELD75] Held, G., et. al., "INGRES: A Relational Data Base System," Proc. 1975 NCC, Anaheim, Ca., May 1975.

[ONG82] Ong, J., "The Design and Implementation of Abstract Data Types in the Relational Database System, INGRES," Masters Report, EECS Dept, University of California, Berkeley, CA Sept. 1982.

[SHOR76] Shortliffe, E., "Computer Based Medical Consultations: MYCIN," Elsevier, New York, 1976.

[STON75] Stonebraker, M., "Implementation of Integrity Constraints and Views by Query Modification," Proc. 1975 ACM-SIGMOD Conference on Management of Data, San Jose, Ca., June 1975.

[STON76] Stonebraker, M. et al., "The Design and Implementation of INGRES," TODS 2, 3, September 1976.

[STON82] Stonebraker, M., et. al., "A Rules System for a Relational Data Base System," Proc. 2nd International Conference on Databases, Jerusalem, Israel, June 1982.

[STON82a] Stonebraker, M., "Extending a DBMS with Added Semantic Knowledge," Proc. NSF Workshop on Data Semantics, Intervale N.H., May 1982. (to appear in Springer-Verlag book edited by M. Brodie)

[STON83] Stonebraker, M., et. al., "Document Processing in a Relational Data Base System," ACM TOOIS, April 1983.

A Fast File System for UNIX*

Revised July 27, 1983

Marshall Kirk McKusick, William N. Joy†,
Samuel J. Leffler‡, Robert S. Fabry

Computer Systems Research Group
Computer Science Division
Department of Electrical Engineering and Computer Science
University of California, Berkeley
Berkeley, CA 94720

ABSTRACT

A reimplementation of the UNIX file system is described. The reimplementation provides substantially higher throughput rates by using more flexible allocation policies, that allow better locality of reference and that can be adapted to a wide range of peripheral and processor characteristics. The new file system clusters data that is sequentially accessed and provides two block sizes to allow fast access for large files while not wasting large amounts of space for small files. File access rates of up to ten times faster than the traditional UNIX file system are experienced. Long needed enhancements to the user interface are discussed. These include a mechanism to lock files, extensions of the name space across file systems, the ability to use arbitrary length file names, and provisions for efficient administrative control of resource usage.

---

* UNIX is a trademark of Bell Laboratories.

†William N. Joy is currently employed by: Sun Microsystems, Inc, 2550 Garcia Avenue, Mountain View, CA 94043

‡Samuel J. Leffler is currently employed by: Lucasfilm Ltd., PO Box 2009, San Rafael, CA 94912

This work was done under grants from the National Science Foundation under grant MCS80-05144, and the Defense Advance Research Projects Agency (DoD) under Arpa Order No. 4031 monitored by Naval Electronic System Command under Contract No. N00039-82-C-0235.

TABLE OF CONTENTS

1. Introduction

This paper describes the changes from the original 512 byte UNIX file system to the new one released with the 4.2 Berkeley Software Distribution. It presents the motivations for the changes, the methods used to affect these changes, the rationale behind the design decisions, and a description of the new implementation. This discussion is followed by a summary of the results that have been obtained, directions for future work, and the additions and changes that have been made to the user visible facilities. The paper concludes with a history of the software engineering of the project.

The original UNIX system that runs on the PDP-11† has simple and elegant file system facilities. File system input/output is buffered by the kernel; there are no alignment constraints on data transfers and all operations are made to appear synchronous. All transfers to the disk are in 512 byte blocks, which can be placed arbitrarily within the data area of the file system. No constraints other than available disk space are placed on file growth [Ritchie74], [Thompson79].

When used on the VAX-11 together with other UNIX enhancements, the original 512 byte UNIX file system is incapable of providing the data throughput rates that many applications require. For example, applications that need to do a small amount of processing on a large quantities of data such as VLSI design and image processing, need to have a high throughput from the file system. High throughput rates are also needed by programs with large address spaces that are constructed by mapping files from the file system into virtual memory. Paging data in and out of the file system is likely to occur frequently. This requires a file system providing higher bandwidth than the original 512 byte UNIX one which provides only about two percent of the maximum disk bandwidth or about 20 kilobytes per second per arm [White80], [Smith81b].

Modifications have been made to the UNIX file system to improve its performance. Since the UNIX file system interface is well understood and not inherently slow, this development retained the abstraction and simply changed the underlying implementation to increase its throughput. Consequently users of the system have not been faced with massive software conversion.

Problems with file system performance have been dealt with extensively in the literature; see [Smith81a] for a survey. The UNIX operating system drew many of its ideas from Multics, a large, high performance operating system [Feiertag71]. Other work includes Hydra [Almes78], Spice [Thompson80], and a file system for a lisp environment [Symbolics81a].

A major goal of this project has been to build a file system that is extensible into a networked environment [Holler73]. Other work on network file systems describe centralized file servers [Accetta80], distributed file servers [Dion80], [Luniewski77], [Porcar82], and protocols to reduce the amount of information that must be transferred across a network [Symbolics81b], [Sturgis80].

† DEC, PDP, VAX, MASSBUS, and UNIBUS are trademarks of Digital Equipment Corporation.

2. Old File System

In the old file system developed at Bell Laboratories each disk drive contains one or more file systems.† A file system is described by its super-block, which contains the basic parameters of the file system. These include the number of data blocks in the file system, a count of the maximum number of files, and a pointer to a list of free blocks. All the free blocks in the system are chained together in a linked list. Within the file system are files. Certain files are distinguished as directories and contain pointers to files that may themselves be directories. Every file has a descriptor associated with it called an *inode*. The inode contains information describing ownership of the file, time stamps marking last modification and access times for the file, and an array of indices that point to the data blocks for the file. For the purposes of this section, we assume that the first 8 blocks of the file are directly referenced by values stored in the inode structure itself*. The inode structure may also contain references to indirect blocks containing further data block indices. In a file system with a 512 byte block size, a singly indirect block contains 128 further block addresses, a doubly indirect block contains 128 addresses of further single indirect blocks, and a triply indirect block contains 128 addresses of further doubly indirect blocks.

A traditional 150 megabyte UNIX file system consists of 4 megabytes of inodes followed by 146 megabytes of data. This organization segregates the inode information from the data; thus accessing a file normally incurs a long seek from its inode to its data. Files in a single directory are not typically allocated slots in consecutive locations in the 4 megabytes of inodes, causing many non-consecutive blocks to be accessed when executing operations on all the files in a directory.

The allocation of data blocks to files is also suboptimum. The traditional file system never transfers more than 512 bytes per disk transaction and often finds that the next sequential data block is not on the same cylinder, forcing seeks between 512 byte transfers. The combination of the small block size, limited read-ahead in the system, and many seeks severely limits file system throughput.

The first work at Berkeley on the UNIX file system attempted to improve both reliability and throughput. The reliability was improved by changing the file system so that all modifications of critical information were staged so that they could either be completed or repaired cleanly by a program after a crash [Kowalski78]. The file system performance was improved by a factor of more than two by changing the basic block size from 512 to 1024 bytes. The increase was because of two factors; each disk transfer accessed twice as much data, and most files could be described without need to access through any indirect blocks since the direct blocks contained twice as much data. The file system with these changes will henceforth be referred to as the *old file system*.

This performance improvement gave a strong indication that increasing the block size was a good method for improving throughput. Although the throughput had doubled, the old file system was still using only about four percent of the disk bandwidth. The main problem was that although the free list was initially ordered for optimal access, it quickly became scrambled as files were created and removed. Eventually the free list became entirely random causing files to have their blocks allocated randomly over the disk. This forced the disk to seek before every block access. Although old file systems provided transfer rates of up to 175 kilobytes per second when they were first created, this rate deteriorated to 30 kilobytes per second after a few weeks of moderate use because of randomization of their free block list. There was no way of restoring the performance an old file system except to dump, rebuild, and restore the file system. Another possibility would be to have a process that periodically reorganized the data on the disk to restore locality as suggested by [Maruyama76].

---

† A file system always resides on a single drive.

* The actual number may vary from system to system, but is usually in the range 5-13.

3. New file system organization

As in the old file system organization each disk drive contains one or more file systems. A file system is described by its super-block, that is located at the beginning of its disk partition. Because the super-block contains critical data it is replicated to protect against catastrophic loss. This is done at the time that the file system is created; since the super-block data does not change, the copies need not be referenced unless a head crash or other hard disk error causes the default super-block to be unusable.

To insure that it is possible to create files as large as 2↑32 bytes with only two levels of indirection, the minimum size of a file system block is 4096 bytes. The size of file system blocks can be any power of two greater than or equal to 4096. The block size of the file system is maintained in the super-block so it is possible for file systems with different block sizes to be accessible simultaneously on the same system. The block size must be decided at the time that the file system is created; it cannot be subsequently changed without rebuilding the file system.

The new file system organization partitions the disk into one or more areas called *cylinder groups*. A cylinder group is comprised of one or more consecutive cylinders on a disk. Associated with each cylinder group is some bookkeeping information that includes a redundant copy of the super-block, space for inodes, a bit map describing available blocks in the cylinder group, and summary information describing the usage of data blocks within the cylinder group. For each cylinder group a static number of inodes is allocated at file system creation time. The current policy is to allocate one inode for each 2048 bytes of disk space, expecting this to be far more than will ever be needed.

All the cylinder group bookkeeping information could be placed at the beginning of each cylinder group. However if this approach were used, all the redundant information would be on the top platter. Thus a single hardware failure that destroyed the top platter could cause the loss of all copies of the redundant super-blocks. Thus the cylinder group bookkeeping information begins at a floating offset from the beginning of the cylinder group. The offset for each successive cylinder group is calculated to be about one track further from the beginning of the cylinder group. In this way the redundant information spirals down into the pack so that any single track, cylinder, or platter can be lost without losing all copies of the super-blocks. Except for the first cylinder group, the space between the beginning of the cylinder group and the beginning of the cylinder group information is used for data blocks.†

3.1. Optimizing storage utilization

Data is laid out so that larger blocks can be transferred in a single disk transfer, greatly increasing file system throughput. As an example, consider a file in the new file system composed of 4096 byte data blocks. In the old file system this file would be composed of 1024 byte blocks. By increasing the block size, disk accesses in the new file system may transfer up to four times as much information per disk transaction. In large files, several 4096 byte blocks may be allocated from the same cylinder so that even larger data transfers are possible before initiating a seek.

The main problem with bigger blocks is that most UNIX file systems are composed of many small files. A uniformly large block size wastes space. Table 1 shows the effect of file system block size on the amount of wasted space in the file system. The machine measured to obtain these figures is one of our time sharing systems that has roughly 1.2 Gigabyte of on-line storage. The measurements are based on the active user file systems containing about 920 megabytes of formated space. The space wasted is measured as the percentage of space on the disk not containing user data. As the block size on the disk increases, the waste rises quickly, to an intolerable 45.6% waste with 4096 byte file system blocks.

† While it appears that the first cylinder group could be laid out with its super-block at the "known" location, this would not work for file systems with blocks sizes of 16K or greater, because of the requirement that the cylinder group information must begin at a block boundary.

Space used	% waste	Organization
775.2 Mb	0.0	Data only, no separation between files
807.8 Mb	4.2	Data only, each file starts on 512 byte boundary
828.7 Mb	6.9	512 byte block UNIX file system
866.5 Mb	11.8	1024 byte block UNIX file system
948.5 Mb	22.4	2048 byte block UNIX file system
1128.3 Mb	45.6	4096 byte block UNIX file system

Table 1 − Amount of wasted space as a function of block size.

To be able to use large blocks without undue waste, small files must be stored in a more efficient way. The new file system accomplishes this goal by allowing the division of a single file system block into one or more *fragments*. The file system fragment size is specified at the time that the file system is created; each file system block can be optionally broken into 2, 4, or 8 fragments, each of which is addressable. The lower bound on the size of these fragments is constrained by the disk sector size, typically 512 bytes. The block map associated with each cylinder group records the space availability at the fragment level; to determine block availability, aligned fragments are examined. Figure 1 shows a piece of a map from a 4096/1024 file system.

Bits in map	XXXX	XXOO	OOXX	OOOO
Fragment numbers	0-3	4-7	8-11	12-15
Block numbers	0	1	2	3

Figure 1 − Example layout of blocks and fragments in a 4096/1024 file system.

Each bit in the map records the status of a fragment; an "X" shows that the fragment is in use, while a "O" shows that the fragment is available for allocation. In this example, fragments 0−5, 10, and 11 are in use, while fragments 6−9, and 12−15 are free. Fragments of adjoining blocks cannot be used as a block, even if they are large enough. In this example, fragments 6−9 cannot be coalesced into a block; only fragments 12−15 are available for allocation as a block.

On a file system with a block size of 4096 bytes and a fragment size of 1024 bytes, a file is represented by zero or more 4096 byte blocks of data, and possibly a single fragmented block. If a file system block must be fragmented to obtain space for a small amount of data, the remainder of the block is made available for allocation to other files. As an example consider an 11000 byte file stored on a 4096/1024 byte file system. This file would uses two full size blocks and a 3072 byte fragment. If no 3072 byte fragments are available at the time the file is created, a full size block is split yielding the necessary 3072 byte fragment and an unused 1024 byte fragment. This remaining fragment can be allocated to another file as needed.

The granularity of allocation is the *write* system call. Each time data is written to a file, the system checks to see if the size of the file has increased*. If the file needs to hold the new data, one of three conditions exists:

1) There is enough space left in an already allocated block to hold the new data. The new data is written into the available space in the block.

2) Nothing has been allocated. If the new data contains more than 4096 bytes, a 4096 byte block is allocated and the first 4096 bytes of new data is written there. This process is repeated until less than 4096 bytes of new data remain. If the remaining new data to be written will fit in three or fewer 1024 byte pieces, an unallocated fragment is located, otherwise a 4096 byte block is located. The new data is written into the located piece.

* A program may be overwriting data in the middle of an existing file in which case space will already be allocated.

3) A fragment has been allocated. If the number of bytes in the new data plus the number of bytes already in the fragment exceeds 4096 bytes, a 4096 byte block is allocated. The contents of the fragment is copied to the beginning of the block and the remainder of the block is filled with the new data. The process then continues as in (2) above. If the number of bytes in the new data plus the number of bytes already in the fragment will fit in three or fewer 1024 byte pieces, an unallocated fragment is located, otherwise a 4096 byte block is located. The contents of the previous fragment appended with the new data is written into the allocated piece.

The problem with allowing only a single fragment on a 4096/1024 byte file system is that data may be potentially copied up to three times as its requirements grow from a 1024 byte fragment to a 2048 byte fragment, then a 3072 byte fragment, and finally a 4096 byte block. The fragment reallocation can be avoided if the user program writes a full block at a time, except for a partial block at the end of the file. Because file systems with different block sizes may coexist on the same system, the file system interface been extended to provide the ability to determine the optimal size for a read or write. For files the optimal size is the block size of the file system on which the file is being accessed. For other objects, such as pipes and sockets, the optimal size is the underlying buffer size. This feature is used by the Standard Input/Output Library, a package used by most user programs. This feature is also used by certain system utilities such as archivers and loaders that do their own input and output management and need the highest possible file system bandwidth.

The space overhead in the 4096/1024 byte new file system organization is empirically observed to be about the same as in the 1024 byte old file system organization. A file system with 4096 byte blocks and 512 byte fragments has about the same amount of space overhead as the 512 byte block UNIX file system. The new file system is more space efficient than the 512 byte or 1024 byte file systems in that it uses the same amount of space for small files while requiring less indexing information for large files. This savings is offset by the need to use more space for keeping track of available free blocks. The net result is about the same disk utilization when the new file systems fragment size equals the old file systems block size.

In order for the layout policies to be effective, the disk cannot be kept completely full. Each file system maintains a parameter that gives the minimum acceptable percentage of file system blocks that can be free. If the the number of free blocks drops below this level only the system administrator can continue to allocate blocks. The value of this parameter can be changed at any time, even when the file system is mounted and active. The transfer rates to be given in section 4 were measured on file systems kept less than 90% full. If the reserve of free blocks is set to zero, the file system throughput rate tends to be cut in half, because of the inability of the file system to localize the blocks in a file. If the performance is impaired because of overfilling, it may be restored by removing enough files to obtain 10% free space. Access speed for files created during periods of little free space can be restored by recreating them once enough space is available. The amount of free space maintained must be added to the percentage of waste when comparing the organizations given in Table 1. Thus, a site running the old 1024 byte UNIX file system wastes 11.8% of the space and one could expect to fit the same amount of data into a 4096/512 byte new file system with 5% free space, since a 512 byte old file system wasted 6.9% of the space.

3.2. File system parameterization

Except for the initial creation of the free list, the old file system ignores the parameters of the underlying hardware. It has no information about either the physical characteristics of the mass storage device, or the hardware that interacts with it. A goal of the new file system is to parameterize the processor capabilities and mass storage characteristics so that blocks can be allocated in an optimum configuration dependent way. Parameters used include the speed of the processor, the hardware support for mass storage transfers, and the characteristics of the mass storage devices. Disk technology is constantly improving and a given installation can have several different disk technologies running on a single processor. Each file system is

parameterized so that it can adapt to the characteristics of the disk on which it is placed.

For mass storage devices such as disks, the new file system tries to allocate new blocks on the same cylinder as the previous block in the same file. Optimally, these new blocks will also be well positioned rotationally. The distance between "rotationally optimal" blocks varies greatly; it can be a consecutive block or a rotationally delayed block depending on system characteristics. On a processor with a channel that does not require any processor intervention between mass storage transfer requests, two consecutive disk blocks often can be accessed without suffering lost time because of an intervening disk revolution. For processors without such channels, the main processor must field an interrupt and prepare for a new disk transfer. The expected time to service this interrupt and schedule a new disk transfer depends on the speed of the main processor.

The physical characteristics of each disk include the number of blocks per track and the rate at which the disk spins. The allocation policy routines use this information to calculate the number of milliseconds required to skip over a block. The characteristics of the processor include the expected time to schedule an interrupt. Given the previous block allocated to a file, the allocation routines calculate the number of blocks to skip over so that the next block in a file will be coming into position under the disk head in the expected amount of time that it takes to start a new disk transfer operation. For programs that sequentially access large amounts of data, this strategy minimizes the amount of time spent waiting for the disk to position itself.

To ease the calculation of finding rotationally optimal blocks, the cylinder group summary information includes a count of the availability of blocks at different rotational positions. Eight rotational positions are distinguished, so the resolution of the summary information is 2 milliseconds for a typical 3600 revolution per minute drive.

The parameter that defines the minimum number of milliseconds between the completion of a data transfer and the initiation of another data transfer on the same cylinder can be changed at any time, even when the file system is mounted and active. If a file system is parameterized to lay out blocks with rotational separation of 2 milliseconds, and the disk pack is then moved to a system that has a processor requiring 4 milliseconds to schedule a disk operation, the throughput will drop precipitously because of lost disk revolutions on nearly every block. If the eventual target machine is known, the file system can be parameterized for it even though it is initially created on a different processor. Even if the move is not known in advance, the rotational layout delay can be reconfigured after the disk is moved so that all further allocation is done based on the characteristics of the new host.

3.3. Layout policies

The file system policies are divided into two distinct parts. At the top level are global policies that use file system wide summary information to make decisions regarding the placement of new inodes and data blocks. These routines are responsible for deciding the placement of new directories and files. They also calculate rotationally optimal block layouts, and decide when to force a long seek to a new cylinder group because there are insufficient blocks left in the current cylinder group to do reasonable layouts. Below the global policy routines are the local allocation routines that use a locally optimal scheme to lay out data blocks.

Two methods for improving file system performance are to increase the locality of reference to minimize seek latency as described by [Trivedi80], and to improve the layout of data to make larger transfers possible as described by [Nevalainen77]. The global layout policies try to improve performance by clustering related information. They cannot attempt to localize all data references, but must also try to spread unrelated data among different cylinder groups. If too much localization is attempted, the local cylinder group may run out of space forcing the data to be scattered to non-local cylinder groups. Taken to an extreme, total localization can result in a single huge cluster of data resembling the old file system. The global policies try to balance the two conflicting goals of localizing data that is concurrently accessed while spreading out unrelated data.

One allocatable resource is inodes. Inodes are used to describe both files and directories. Files in a directory are frequently accessed together. For example the "list directory" command often accesses the inode for each file in a directory. The layout policy tries to place all the files in a directory in the same cylinder group. To ensure that files are allocated throughout the disk, a different policy is used for directory allocation. A new directory is placed in the cylinder group that has a greater than average number of free inodes, and the fewest number of directories in it already. The intent of this policy is to allow the file clustering policy to succeed most of the time. The allocation of inodes within a cylinder group is done using a next free strategy. Although this allocates the inodes randomly within a cylinder group, all the inodes for each cylinder group can be read with 4 to 8 disk transfers. This puts a small and constant upper bound on the number of disk transfers required to access all the inodes for all the files in a directory as compared to the old file system where typically, one disk transfer is needed to get the inode for each file in a directory.

The other major resource is the data blocks. Since data blocks for a file are typically accessed together, the policy routines try to place all the data blocks for a file in the same cylinder group, preferably rotationally optimally on the same cylinder. The problem with allocating all the data blocks in the same cylinder group is that large files will quickly use up available space in the cylinder group, forcing a spill over to other areas. Using up all the space in a cylinder group has the added drawback that future allocations for any file in the cylinder group will also spill to other areas. Ideally none of the cylinder groups should ever become completely full. The solution devised is to redirect block allocation to a newly chosen cylinder group when a file exceeds 32 kilobytes, and at every megabyte thereafter. The newly chosen cylinder group is selected from those cylinder groups that have a greater than average number of free blocks left. Although big files tend to be spread out over the disk, a megabyte of data is typically accessible before a long seek must be performed, and the cost of one long seek per megabyte is small.

The global policy routines call local allocation routines with requests for specific blocks. The local allocation routines will always allocate the requested block if it is free. If the requested block is not available, the allocator allocates a free block of the requested size that is rotationally closest to the requested block. If the global layout policies had complete information, they could always request unused blocks and the allocation routines would be reduced to simple bookkeeping. However, maintaining complete information is costly; thus the implementation of the global layout policy uses heuristic guesses based on partial information.

If a requested block is not available the local allocator uses a four level allocation strategy:

1) Use the available block rotationally closest to the requested block on the same cylinder.
2) If there are no blocks available on the same cylinder, use a block within the same cylinder group.
3) If the cylinder group is entirely full, quadratically rehash among the cylinder groups looking for a free block.
4) Finally if the rehash fails, apply an exhaustive search.

The use of quadratic rehash is prompted by studies of symbol table strategies used in programming languages. File systems that are parameterized to maintain at least 10% free space almost never use this strategy; file systems that are run without maintaining any free space typically have so few free blocks that almost any allocation is random. Consequently the most important characteristic of the strategy used when the file system is low on space is that it be fast.

4. Performance

Ultimately, the proof of the effectiveness of the algorithms described in the previous section is the long term performance of the new file system.

Our empiric studies have shown that the inode layout policy has been effective. When running the "list directory" command on a large directory that itself contains many directories, the number of disk accesses for inodes is cut by a factor of two. The improvements are even more dramatic for large directories containing only files, disk accesses for inodes being cut by a factor of eight. This is most encouraging for programs such as spooling daemons that access many small files, since these programs tend to flood the disk request queue on the old file system.

Table 2 summarizes the measured throughput of the new file system. Several comments need to be made about the conditions under which these tests were run. The test programs measure the rate that user programs can transfer data to or from a file without performing any processing on it. These programs must write enough data to insure that buffering in the operating system does not affect the results. They should also be run at least three times in succession; the first to get the system into a known state and the second two to insure that the experiment has stabilized and is repeatable. The methodology and test results are discussed in detail in [Kridle83]†. The systems were running multi-user but were otherwise quiescent. There was no contention for either the cpu or the disk arm. The only difference between the UNIBUS and MASSBUS tests was the controller. All tests used an Ampex Capricorn 330 Megabyte Winchester disk. As Table 2 shows, all file system test runs were on a VAX 11/750. All file systems had been in production use for at least a month before being measured.

Type of File System	Processor and Bus Measured	Read Speed	Read Bandwidth	Read % CPU
old 1024	750/UNIBUS	29 Kbytes/sec	29/1100 3%	11%
new 4096/1024	750/UNIBUS	221 Kbytes/sec	221/1100 20%	43%
new 8192/1024	750/UNIBUS	233 Kbytes/sec	233/1100 21%	29%
new 4096/1024	750/MASSBUS	466 Kbytes/sec	466/1200 39%	73%
new 8192/1024	750/MASSBUS	466 Kbytes/sec	466/1200 39%	54%

Table 2a – Reading rates of the old and new UNIX file systems.

Type of File System	Processor and Bus Measured	Write Speed	Write Bandwidth	Write % CPU
old 1024	750/UNIBUS	48 Kbytes/sec	48/1100 4%	29%
new 4096/1024	750/UNIBUS	142 Kbytes/sec	142/1100 13%	43%
new 8192/1024	750/UNIBUS	215 Kbytes/sec	215/1100 19%	46%
new 4096/1024	750/MASSBUS	323 Kbytes/sec	323/1200 27%	94%
new 8192/1024	750/MASSBUS	466 Kbytes/sec	466/1200 39%	95%

Table 2b – Writing rates of the old and new UNIX file systems.

Unlike the old file system, the transfer rates for the new file system do not appear to change over time. The throughput rate is tied much more strongly to the amount of free space that is maintained. The measurements in Table 2 were based on a file system run with 10% free space. Synthetic work loads suggest the performance deteriorates to about half the throughput rates given in Table 2 when no free space is maintained.

The percentage of bandwidth given in Table 2 is a measure of the effective utilization of the disk by the file system. An upper bound on the transfer rate from the disk is measured by doing 65536* byte reads from contiguous tracks on the disk. The bandwidth is calculated by

† A UNIX command that is similar to the reading test that we used is, "cp file /dev/null", where "file" is eight Megabytes long.

* This number, 65536, is the maximal I/O size supported by the VAX hardware. it is a remnant of the

comparing the data rates the file system is able to achieve as a percentage of this rate. Using this metric, the old file system is only able to use about 3-4% of the disk bandwidth, while the new file system uses up to 39% of the bandwidth.

In the new file system, the reading rate is always at least as fast as the writing rate. This is to be expected since the kernel must do more work when allocating blocks than when simply reading them. Note that the write rates are about the same as the read rates in the 8192 byte block file system; the write rates are slower than the read rates in the 4096 byte block file system. The slower write rates occur because the kernel has to do twice as many disk allocations per second, and the processor is unable to keep up with the disk transfer rate.

In contrast the old file system is about 50% faster at writing files than reading them. This is because the *write* system call is asynchronous and the kernel can generate disk transfer requests much faster than they can be serviced, hence disk transfers build up in the disk buffer cache. Because the disk buffer cache is sorted by minimum seek order, the average seek between the scheduled disk writes is much less than they would be if the data blocks are written out in the order in which they are generated. However when the file is read, the *read* system call is processed synchronously so the disk blocks must be retrieved from the disk in the order in which they are allocated. This forces the disk scheduler to do long seeks resulting in a lower throughput rate.

The performance of the new file system is currently limited by a memory to memory copy operation because it transfers data from the disk into buffers in the kernel address space and then spends 40% of the processor cycles copying these buffers to user address space. If the buffers in both address spaces are properly aligned, this transfer can be affected without copying by using the VAX virtual memory management hardware. This is especially desirable when large amounts of data are to be transferred. We did not implement this because it would change the semantics of the file system in two major ways; user programs would be required to allocate buffers on page boundaries, and data would disappear from buffers after being written.

Greater disk throughput could be achieved by rewriting the disk drivers to chain together kernel buffers. This would allow files to be allocated to contiguous disk blocks that could be read in a single disk transaction. Most disks contain either 32 or 48 512 byte sectors per track. The inability to use contiguous disk blocks effectively limits the performance on these disks to less than fifty percent of the available bandwidth. Since each track has a multiple of sixteen sectors it holds exactly two or three 8192 byte file system blocks, or four or six 4096 byte file system blocks. If the the next block for a file cannot be laid out contiguously, then the minimum spacing to the next allocatable block on any platter is between a sixth and a half a revolution. The implication of this is that the best possible layout without contiguous blocks uses only half of the bandwidth of any given track. If each track contains an odd number of sectors, then it is possible to resolve the rotational delay to any number of sectors by finding a block that begins at the desired rotational position on another track. The reason that block chaining has not been implemented is because it would require rewriting all the disk drivers in the system, and the current throughput rates are already limited by the speed of the available processors.

Currently only one block is allocated to a file at a time. A technique used by the DEMOS file system when it finds that a file is growing rapidly, is to preallocate several blocks at once, releasing them when the file is closed if they remain unused. By batching up the allocation the system can reduce the overhead of allocating at each write, and it can cut down on the number of disk writes needed to keep the block pointers on the disk synchronized with the block allocation [Powell79].

---

system's PDP-11 ancestry.

5. File system functional enhancements

The speed enhancements to the UNIX file system did not require any changes to the semantics or data structures viewed by the users. However several changes have been generally desired for some time but have not been introduced because they would require users to dump and restore all their file systems. Since the new file system already requires that all existing file systems be dumped and restored, these functional enhancements have been introduced at this time.

5.1. Long file names

File names can now be of nearly arbitrary length. The only user programs affected by this change are those that access directories. To maintain portability among UNIX systems that are not running the new file system, a set of directory access routines have been introduced that provide a uniform interface to directories on both old and new systems.

Directories are allocated in units of 512 bytes. This size is chosen so that each allocation can be transferred to disk in a single atomic operation. Each allocation unit contains variable-length directory entries. Each entry is wholly contained in a single allocation unit. The first three fields of a directory entry are fixed and contain an inode number, the length of the entry, and the length of the name contained in the entry. Following this fixed size information is the null terminated name, padded to a 4 byte boundary. The maximum length of a name in a directory is currently 255 characters.

Free space in a directory is held by entries that have a record length that exceeds the space required by the directory entry itself. All the bytes in a directory unit are claimed by the directory entries. This normally results in the last entry in a directory being large. When entries are deleted from a directory, the space is returned to the previous entry in the same directory unit by increasing its length. If the first entry of a directory unit is free, then its inode number is set to zero to show that it is unallocated.

5.2. File locking

The old file system had no provision for locking files. Processes that needed to synchronize the updates of a file had to create a separate "lock" file to synchronize their updates. A process would try to create a "lock" file. If the creation succeeded, then it could proceed with its update; if the creation failed, then it would wait, and try again. This mechanism had three drawbacks. Processes consumed CPU time, by looping over attempts to create locks. Locks were left lying around following system crashes and had to be cleaned up by hand. Finally, processes running as system administrator are always permitted to create files, so they had to use a different mechanism. While it is possible to get around all these problems, the solutions are not straight-forward, so a mechanism for locking files has been added.

The most general schemes allow processes to concurrently update a file. Several of these techniques are discussed in [Peterson83]. A simpler technique is to simply serialize access with locks. To attain reasonable efficiency, certain applications require the ability to lock pieces of a file. Locking down to the byte level has been implemented in the Onyx file system by [Bass81]. However, for the applications that currently run on the system, a mechanism that locks at the granularity of a file is sufficient.

Locking schemes fall into two classes, those using hard locks and those using advisory locks. The primary difference between advisory locks and hard locks is the decision of when to override them. A hard lock is always enforced whenever a program tries to access a file; an advisory lock is only applied when it is requested by a program. Thus advisory locks are only effective when all programs accessing a file use the locking scheme. With hard locks there must be some override policy implemented in the kernel, with advisory locks the policy is implemented by the user programs. In the UNIX system, programs with system administrator privilege can override any protection scheme. Because many of the programs that need to use locks run as system administrators, we chose to implement advisory locks rather than create a

protection scheme that was contrary to the UNIX philosophy or could not be used by system administration programs.

The file locking facilities allow cooperating programs to apply advisory *shared* or *exclusive* locks on files. Only one process has an exclusive lock on a file while multiple shared locks may be present. Both shared and exclusive locks cannot be present on a file at the same time. If any lock is requested when another process holds an exclusive lock, or an exclusive lock is requested when another process holds any lock, the open will block until the lock can be gained. Because shared and exclusive locks are advisory only, even if a process has obtained a lock on a file, another process can override the lock by opening the same file without a lock.

Locks can be applied or removed on open files, so that locks can be manipulated without needing to close and reopen the file. This is useful, for example, when a process wishes to open a file with a shared lock to read some information, to determine whether an update is required. It can then get an exclusive lock so that it can do a read, modify, and write to update the file in a consistent manner.

A request for a lock will cause the process to block if the lock can not be immediately obtained. In certain instances this is unsatisfactory. For example, a process that wants only to check if a lock is present would require a separate mechanism to find out this information. Consequently, a process may specify that its locking request should return with an error if a lock can not be immediately obtained. Being able to poll for a lock is useful to "daemon" processes that wish to service a spooling area. If the first instance of the daemon locks the directory where spooling takes place, later daemon processes can easily check to see if an active daemon exists. Since the lock is removed when the process exits or the system crashes, there is no problem with unintentional locks files that must be cleared by hand.

Almost no deadlock detection is attempted. The only deadlock detection made by the system is that the file descriptor to which a lock is applied does not currently have a lock of the same type (i.e. the second of two successive calls to apply a lock of the same type will fail). Thus a process can deadlock itself by requesting locks on two separate file descriptors for the same object.

5.3. Symbolic links

The 512 byte UNIX file system allows multiple directory entries in the same file system to reference a single file. The link concept is fundamental; files do not live in directories, but exist separately and are referenced by links. When all the links are removed, the file is deallocated. This style of links does not allow references across physical file systems, nor does it support inter-machine linkage. To avoid these limitations *symbolic links* have been added similar to the scheme used by Multics [Feiertag71].

A symbolic link is implemented as a file that contains a pathname. When the system encounters a symbolic link while interpreting a component of a pathname, the contents of the symbolic link is prepended to the rest of the pathname, and this name is interpreted to yield the resulting pathname. If the symbolic link contains an absolute pathname, the absolute pathname is used, otherwise the contents of the symbolic link is evaluated relative to the location of the link in the file hierarchy.

Normally programs do not want to be aware that there is a symbolic link in a pathname that they are using. However certain system utilities must be able to detect and manipulate symbolic links. Three new system calls provide the ability to detect, read, and write symbolic links, and seven system utilities were modified to use these calls.

In future Berkeley software distributions it will be possible to mount file systems from other machines within a local file system. When this occurs, it will be possible to create symbolic links that span machines.

5.4. Rename

Programs that create new versions of data files typically create the new version as a temporary file and then rename the temporary file with the original name of the data file. In the old UNIX file systems the renaming required three calls to the system. If the program were interrupted or the system crashed between these calls, the data file could be left with only its temporary name. To eliminate this possibility a single system call has been added that performs the rename in an atomic fashion to guarantee the existence of the original name.

In addition, the rename facility allows directories to be moved around in the directory tree hierarchy. The rename system call performs special validation checks to insure that the directory tree structure is not corrupted by the creation of loops or inaccessible directories. Such corruption would occur if a parent directory were moved into one of its descendants. The validation check requires tracing the ancestry of the target directory to insure that it does not include the directory being moved.

5.5. Quotas

The UNIX system has traditionally attempted to share all available resources to the greatest extent possible. Thus any single user can allocate all the available space in the file system. In certain environments this is unacceptable. Consequently, a quota mechanism has been added for restricting the amount of file system resources that a user can obtain. The quota mechanism sets limits on both the number of files and the number of disk blocks that a user may allocate. A separate quota can be set for each user on each file system. Each resource is given both a hard and a soft limit. When a program exceeds a soft limit, a warning is printed on the users terminal; the offending program is not terminated unless it exceeds its hard limit. The idea is that users should stay below their soft limit between login sessions, but they may use more space while they are actively working. To encourage this behavior, users are warned when logging in if they are over any of their soft limits. If they fail to correct the problem for too many login sessions, they are eventually reprimanded by having their soft limit enforced as their hard limit.

6. Software engineering

The preliminary design was done by Bill Joy in late 1980; he presented the design at The USENIX Conference held in San Francisco in January 1981. The implementation of his design was done by Kirk McKusick in the summer of 1981. Most of the new system calls were implemented by Sam Leffler. The code for enforcing quotas was implemented by Robert Elz at the University of Melbourne.

To understand how the project was done it is necessary to understand the interfaces that the UNIX system provides to the hardware mass storage systems. At the lowest level is a *raw disk*. This interface provides access to the disk as a linear array of sectors. Normally this interface is only used by programs that need to do disk to disk copies or that wish to dump file systems. However, user programs with proper access rights can also access this interface. A disk is usually formated with a file system that is interpreted by the UNIX system to provide a directory hierarchy and files. The UNIX system interprets and multiplexes requests from user programs to create, read, write, and delete files by allocating and freeing inodes and data blocks. The interpretation of the data on the disk could be done by the user programs themselves. The reason that it is done by the UNIX system is to synchronize the user requests, so that two processes do not attempt to allocate or modify the same resource simultaneously. It also allows access to be restricted at the file level rather than at the disk level and allows the common file system routines to be shared between processes.

The implementation of the new file system amounted to using a different scheme for formating and interpreting the disk. Since the synchronization and disk access routines themselves were not being changed, the changes to the file system could be developed by moving the file system interpretation routines out of the kernel and into a user program. Thus, the first step was to extract the file system code for the old file system from the UNIX kernel and change its requests to the disk driver to accesses to a raw disk. This produced a library of routines that mapped what would normally be system calls into read or write operations on the raw disk. This library was then debugged by linking it into the system utilities that copy, remove, archive, and restore files.

A new cross file system utility was written that copied files from the simulated file system to the one implemented by the kernel. This was accomplished by calling the simulation library to do a read, and then writing the resultant data by using the conventional write system call. A similar utility copied data from the kernel to the simulated file system by doing a conventional read system call and then writing the resultant data using the simulated file system library.

The second step was to rewrite the file system simulation library to interpret the new file system. By linking the new simulation library into the cross file system copying utility, it was possible to easily copy files from the old file system into the new one and from the new one to the old one. Having the file system interpretation implemented in user code had several major benefits. These included being able to use the standard system tools such as the debuggers to set breakpoints and single step through the code. When bugs were discovered, the offending problem could be fixed and tested without the need to reboot the machine. There was never a period where it was necessary to maintain two concurrent file systems in the kernel. Finally it was not necessary to dedicate a machine entirely to file system development, except for a brief period while the new file system was boot strapped.

The final step was to merge the new file system back into the UNIX kernel. This was done in less than two weeks, since the only bugs remaining were those that involved interfacing to the synchronization routines that could not be tested in the simulated system. Again the simulation system proved useful since it enabled files to be easily copied between old and new file systems regardless of which file system was running in the kernel. This greatly reduced the number of times that the system had to be rebooted.

The total design and debug time took about one man year. Most of the work was done on the file system utilities, and changing all the user programs to use the new facilities. The code changes in the kernel were minor, involving the addition of only about 800 lines of code

(including comments).

Acknowledgements

We thank Robert Elz for his ongoing interest in the new file system, and for adding disk quotas in a rational and efficient manner. We also acknowledge Dennis Ritchie for his suggestions on the appropriate modifications to the user interface. We appreciate Michael Powell's explanations on how the DEMOS file system worked; many of his ideas were used in this implementation. Special commendation goes to Peter Kessler and Robert Henry for acting like real users during the early debugging stage when files were less stable than they should have been. Finally we thank our sponsors, the National Science Foundation under grant MCS80-05144, and the Defense Advance Research Projects Agency (DoD) under Arpa Order No. 4031 monitored by Naval Electronic System Command under Contract No. N00039-82-C-0235.

References

[Accetta80] Accetta, M., Robertson, G., Satyanarayanan, M., and Thompson, M. "The Design of a Network-Based Central File System", Carnegie-Mellon University, Dept of Computer Science Tech Report, #CMU-CS-80-134

[Almes78] Almes, G., and Robertson, G. "An Extensible File System for Hydra" Proceedings of the Third International Conference on Software Engineering, IEEE, May 1978.

[Bass81] Bass, J. "Implementation Description for File Locking", Onyx Systems Inc, 73 E. Trimble Rd, San Jose, CA 95131 Jan 1981.

[Dion80] Dion, J. "The Cambridge File Server", Operating Systems Review, 14, 4. Oct 1980. pp 26-35

[Eswaran74] Eswaran, K. "Placement of records in a file and file allocation in a computer network", Proceedings IFIPS, 1974. pp 304-307

[Holler73] Holler, J. "Files in Computer Networks", First European Workshop on Computer Networks, April 1973. pp 381-396

[Feiertag71] Feiertag, R. J. and Organick, E. I., "The Multics Input-Output System", *Proceedings of the Third Symposium on Operating Systems Principles*, ACM, Oct 1971. pp 35-41

[Kridle83] Kridle, R., and McKusick, M., "Performance Effects of Disk Subsystem Choices for VAX Systems Running 4.2BSD UNIX", Computer Systems Research Group, Dept of EECS, Berkeley, CA 94720, Technical Report #8.

[Kowalski78] Kowalski, T. "FSCK - The UNIX System Check Program", Bell Laboratory, Murray Hill, NJ 07974. March 1978

[Luniewski77] Luniewski, A. "File Allocation in a Distributed System", MIT Laboratory for Computer Science, Dec 1977.

[Maruyama76] Maruyama, K., and Smith, S. "Optimal reorganization of Distributed Space Disk Files", Communications of the ACM, 19, 11. Nov 1976. pp 634-642

[Nevalainen77] Nevalainen, O., Vesterinen, M. "Determining Blocking Factors for Sequential Files by Heuristic Methods", The Computer Journal, 20, 3. Aug 1977. pp 245-247

[Peterson83] Peterson, G. "Concurrent Reading While Writing", ACM Transactions on Programming Languages and Systems, ACM, 5, 1. Jan 1983. pp 46-55

[Powell79] Powell, M. "The DEMOS File System", Proceedings of the Sixth Symposium on Operating Systems Principles, ACM, Nov 1977. pp 33-42

[Porcar82] Porcar, J. "File Migration in Distributed Computer Systems", Ph.D. Thesis, Lawrence Berkeley Laboratory Tech Report #LBL-14763.

[Ritchie74] Ritchie, D. M. and Thompson, K., "The UNIX Time-Sharing System", CACM 17, 7. July 1974. pp 365-375

[Smith81a] Smith, A. "*Input/Output Optimization and Disk Architectures*: A Survey", Performance and Evaluation 1. Jan 1981. pp 104-117

[Smith81b] Smith, A. "Bibliography on File and I/O System Optimization and Related Topics", Operating Systems Review, 15, 4. Oct 1981. pp 39-54

[Sturgis80] Sturgis, H., Mitchell, J., and Israel, J. "Issues in the Design and Use of a Distributed File System", Operating Systems Review, 14, 3. pp 55-79

[Symbolics81a] "Symbolics File System", Symbolics Inc, 9600 DeSoto Ave, Chatsworth, CA 91311 Aug 1981.

[Symbolics81b] "Chaosnet FILE Protocol". Symbolics Inc, 9600 DeSoto Ave, Chatsworth, CA 91311 Sept 1981.

[*Thompson*79] Thompson, K. "UNIX Implementation", Section 31, Volume 2B, UNIX Programmers Manual, Bell Laboratory, Murray Hill, NJ 07974. Jan 1979

[Thompson80] Thompson, M. "Spice File System", Carnegie-Mellon University, Dept of Computer Science Tech Report, #CMU-CS-80-???

[Trivedi80] Trivedi, K. "Optimal Selection of CPU Speed, Device Capabilities, and File Assignments", Journal of the ACM, 27, 3. July 1980. pp 457-473

[White80] White, R. M. "Disk Storage Technology", Scientific American, 243(2), August 1980.

Finding Files Fast

James A. Woods

Informatics General Corporation
NASA Ames Research Center
Moffett Field, California 94035

January 15, 1983

ABSTRACT

A fast filename search facility for UNIX is presented. It consolidates two data compression methods with a novel string search technique to rapidly locate arbitrary files. The code, integrated into the standard *find* utility, consults a preprocessed database, regenerated daily. This contrasts with the usual mechanism of matching search keys against candidate items generated on-the-fly from a scattered directory structure.

The pathname database is an incrementally-encoded lexicographically sorted list (sometimes referred to as a "front-compressed" file) which is also subjected to common bigram coding to effect further space reduction. The storage savings are a factor of five to six over the standard ascii representation. The list is scanned using a modified linear search specially tailored to the incremental encoding; typical "user time" required by this algorithm is 40%-50% less than with naive search.

Introduction

Locating files in a computer system, or network of systems, is a common activity. UNIX users have recourse to a variety of approaches, ranging from manipulation of *cd*, *ls*, and *grep* commands, to specialized programs such as U. C. Berkeley's *whereis* and *fleece*, to the more general UNIX *find*.

The Berkeley *fleece* is unfortunately restricted to home directories, and *whereis* is limited to eke-ing out system code/documentation residing in standard places. The arbitrary

```
find / -name "*<filename>*" -print
```

will certainly locate files when the associated directory structure cannot be recalled, but is inherently slow as it recursively descends the entire file system to mercilessly thrash about the disk. Impatience has prompted us to develop an alternative to the "seek and ye shall find" method of pathname search.

Precomputation

Why not simply build a static list of all files on the system to search with *grep*? Alas, a healthy system with 20000 files contains upwards of 1000 blocks of filenames, even with an abbreviated */u* (vs. */usr*) adopted for user home prefixes. *Grep* on our unloaded 30-40 block/second PDP 11/70 system demands half a minute for the scan. This is unacceptable for an oft-used command.

Incidently, it is not much of a sacrifice to be unable to reference files which are less than a day old—either the installer is likely to be contactable, or the file is not quite ready for use! Well-aged files originated by other groups, usually with different filesystem naming conventions, are the probable candidates for search.

Compression

To speed access for the application, one might consider binary search or hashing, but these schemes do not work well for partial matching, where we are interested in portions of pathnames. Though fast, the methods do not save space, which is often at a premium. An easily implementable

space saving technique for ordered data, known as incremental encoding, has been adapted for the similar task of dictionary compression [Morris/Thompson, 1974]. Here, a count of the longest prefix of the preceding name is computed. For example,

```
/usr/src
/usr/src/cmd/aardvark.c
/usr/src/cmd/armadillo.c
/usr/tmp/zoo
```

transforms to

```
 0 /usr/src
 8 /cmd/aardvark.c
14 armadillo.c
 5 tmp/zoo
```

If we choose to delimit the pathname residue with parity-marked count bytes, decoding can be as simple as (omitting declarations):

```
fp = fopen ( COMPRESSED_FILELIST, "r" );
while ( (count = (getc ( fp ) & 0177)) != EOF ) {
        for ( p = path + count; (*p++ = getc ( fp )) < 0200; )
                ;                       /* overlay old path with new */
        ungetc ( *--p, fp );
        *p-- = NULL;
        if ( match ( path, name ) == YES )
                puts ( path );
}
```

where *match* is a favorite routine to determine if string *path* contains *name*.

In fact, since the coded filelist is about five times shorter than the uncoded one, and the decoding is very easy, this program runs about three to four times as fast as the efficient *grep* on the expanded file.

Speedier Yet

Useful as it is, there is still room for improvement. (Aside: this code is best inserted into the distributed *find*. There is no need to burden UNIX with another command [and manual page] when we can improve an existing similar program. Conveniently, there is no two-argument form of *find* so we can fill the vacuum with an unadorned

```
find name
```

to perform the function.)

Notice that the above code fragment still searches through all the characters of expanded list, albeit in main memory instead of disk. It turns out that this can be avoided by matching the name substring ***backwards*** against a reversed pathname, until the boundary delineated by the repetition count. Assuming *namend* points to the final character of a NULL-byte prefixed *name*, then replace *match* by

```
for ( s = p, cutoff = path + count; s >= cutoff; s-- ) {
        if ( *s == *namend ) {          /* quick first char check */
                for ( p = namend - 1, q = s - 1; *p != NULL; p--, q-- )
                        if ( *q != *p )
                                break;
                if ( *p == NULL ) {
                        puts ( path );
                        break;
                }
        }
}
```

This is more easily understood by considering three cases. If the substring lies wholly to the right of the cutoff, the match will terminate successfully. If there is an overlap, the cutoff becomes "soft" and the match continues. If the substring lies completely to the left of the cutoff, then a match would have been discovered for an earlier pathname, so we need not search these characters! Technically, *cutoff* must be re-anchored to *path* immediately after matches. This condition is omitted above for the sake of clarity. Statistics on overlap have not been garnered, but a 40-50% speedup is consistently observed.

The author has not discovered this refinement in the literature.

Two Tier Technique

Shell-style filename expansion without undue slowdown can be had by first performing the fast search on a metacharacter-free component of *name*, then applying regular expression syntax "globbing" to these selected paths via the slower recursive *amatch* function internal to *find*. Ergo,

```
puts ( path );
```

becomes

```
if ( globchars == NO | amatch ( path, name ) )
        puts ( path );
```

where *globchars* is set if *name* contains shell glob characters. Using wildcarding, a primitive *man* command might be

```
vtroff -man 'find '*man*'"$1".[1-9]'
```

Diminishing Returns

Production *find* code at Ames exacts a further 20-25% space compression (entropy reduction) by assigning single non-printing ascii codes to the most common 128 bigrams. ".c" and ".P" figure prominently. Room for these codes is made by reserving only 28 count codes for the likeliest "differential" counts (the interline difference between one prefix count and the next), along with a "switch" code for out-of-range counts (remember the possible 1024 byte pathnames, courtesy BSD 4.2). Printable ascii comprises the filename residue. We will not dwell on this rather *ad hoc* means, which barely reduces search time.

Other algorithms to address the time-space complexity tradeoff such as Huffman or restricted variability coding [Reghbati, 1981] do not look promising—they only change an I/O-bound process to a compute-bound one. Some experiments were done with the inverted file programs *inv* and *hunt*. Here, process startup overhead (the *fgrep* call to disambiguate "false drops") and space consumption (full pathnames plus an index) make *inv* invocations noncompetitive. Boyer-Moore sublinear search [Boyer, 1977] or macro model methods [Storer/Szymanski, 1982] might be employed, but must concern typically short 4-10 character patterns and equally short post-compression pathname content, for all their added complexity.

To conclude, we are content to scan 19000 filenames in several seconds using 180 blocks and two extra pages of C code.

REFERENCES

Boyer, R. S. *A Fast String Searching Algorithm*, Commun. ACM, Vol. 20, No. 10, October 1977.

Morris, R. and Thompson, K. *Webster's Second on the Head of a Pin*, Unpublished Technical Memo, Bell Laboratories, Murray Hill, N. J., 1974.

Reghbati, H. K. *An Overview of Data Compression Techniques*, Computer, Vol. 14, No. 4, April 1981.

Storer, J. A. and Szymanski, T. G. *Data Compression via Textual Substitution*, J. ACM, Vol. 29, No. 4, October 1982.

JAMES A. WOODS

3740 25th Street #606
San Francisco, California 94110

EDUCATION

University of California, Berkeley

MS Electrical Engineering and Computer Sciences, 1975
Thesis: *Digital Data Compression of Music with emphasis on Orthogonal Transform Coding*

AB Computer Science, 1974
Fannie and John Hertz Scholarship, 1971-1974

EXPERIENCE

6/77-present Systems Programmer/Analyst, Informatics General, contractor to NASA Ames Research Center, Moffett Field, California. Systems/applications support, installation management, and consulting for users of several machines involved in a variety of scientific projects. Analysis/specification of software and hardware requirements for various design studies in the realm of man-vehicle systems.

UNIX-related work includes:

U. C. Berkeley software installation/adaptation (4.2 and 2.8 BSD), v6/v7 conversion, file system maintenance and reconfiguration. Responsible for Apple/UNIX file transfer software, general utility development and associated algorithm design (tape file archival, enhanced spelling correctors, etc.). Involved with statistical/graphical data analysis for aircraft human factors research, support for cross-machine systems development, tutoring of both naive and sophisticated users in the subjects of text processing, C, shell, and awk programming.

Xerox Sigma 9 (32-bit mainframe) efforts:

System generation, administration, and customer engineer interface. Monitor modification and performance tuning. Software support for real-time aircraft flight simulations, including coding for real-time servicing of cockpit instrumentation, diagnostic programming for analog/digital and communication link hardware, real-time library design/development. Installation and customization of packages such as LBL's Software Tools, UCLA's BMDP. Time-critical support for PIONEER/VENUS mission satellite data reduction.

6/75-12/75 Graduate work, U. C. Berkeley. Coursework topics in theoretical computer science, compiler construction, information theory, etc. Project work in signal processing and psychoacoustics. Implemented Fast Fourier Transform routines for a non-floating-point machine running UNIX.

7/74-3/75 Undergraduate employment, U. C. Berkeley. Various computer-related positions under aegis of work-study program, including linear programming for the U. S. Forest Service, and Snobol coding to correlate colonial shipping trade routes for the history department.

PROFESSIONAL INTERESTS

Member ACM, IEEE Computer Society. Subscribe to nearly twenty technical journals. Avid follower of progress in computer science theory, digital audio, raster graphics, text searching, functional programming, and mathematical

typography. Consider data compression of information (audio/video signals, text) a vital research area. Developer of fast anagram solution methods. Inventor of a novel string search technique for compressed files. Unabashed UNIX fan.

PUBLICATIONS

Finding Files Fast, ;login:, February/March 1983.

PERSONAL

Born 1953. Appreciate book and wine collecting, computer chess, new wave and folk music. Five ball juggler.

DATABASE PORTALS: A NEW APPLICATION PROGRAM INTERFACE

by

Michael Stonebraker and Lawrence A. Rowe

Memorandum No. UCB/ERL M82/80

2 November 1982

ELECTRONICS RESEARCH LABORATORY

DATABASE PORTALS: A NEW APPLICATION PROGRAM INTERFACE

Michael Stonebraker
Lawrence A. Rowe

Department of Electrical Engineering and Computer Science
University of California
Berkeley, CA 94720

ABSTRACT

This paper describes the design and one proposed implementation of a new application program interface to a database management system. Programs which browse through a database making ad-hoc updates are not well served by conventional embeddings of DBMS commands in programming languages. A new embedding is suggested which overcomes all deficiencies. This construct, called a *portal*, allows a program to request a collection of tuples at once and supports novel concurrency control schemes.

November 2, 1982

DATABASE PORTALS: A NEW APPLICATION PROGRAM INTERFACE

Michael Stonebraker
Lawrence A. Rowe

Department of Electrical Engineering and Computer Science
University of California
Berkeley, CA 94720

1. INTRODUCTION

There have been several recent proposals for user interfaces whereby a person can "browse" through a database [CATE80, HER080, MARY80, ROWE82, STON82, ZLOO82]. Such interfaces allow one to select data of interest (e.g., "all employees over 40") and then navigate through this data making ad-hoc changes.

A simple illustration of a browsing program is described with the aid of figure 1. This program allows a user to "edit" a relation. It is similar to a full screen, visual text editor (e.g., vi [JOY79] or EMACS [STAL81]) except that a relation is edited rather than a text file. This example browser will be used to motivate the need for a new programming language interface to a database management system.

In figure 1 data from an *employee* relation is displayed. Since only a few rows of the relation can fit on the screen at one time, cursor commands are provided to scroll forward and backward. In other words, the screen provides a "portal" onto the employee relation which the user can reposition. Commands are also provided so a user can edit the data on the screen. For example, Dave Smith's salary can be changed by repositioning the cursor to the field containing

employee relation

name	age	salary	dept
Ken Johnson	43	25000	sales
Sue Keller	40	28000	accounting
Dave Smith	52	30000	purchasing
Kathy Able	28	22000	accounting
George Toms	26	18000	shipping
Mike Baker	34	27000	sales

find insert delete update quit

Figure 1. Relation editor interface.

30,000 and entering a new value.

Other operations are listed at the bottom of figure 1. The *find* operation scans forward or backward through the data from the row the CRT cursor is on until the first row is found that satisfies a user specified predicate. The *insert* and *delete* operations allow the user to enter or remove rows from the table. The *update* operation commits changes to the database so they become visible to other users. Lastly, the *quit* operation exits the editor.

The data manipulation facilities supported by conventional programming language interfaces [ALLM76, ASTR76, SCHM77, ROWE79, WASS79] allow a program to bind a query to a database cursor,[1] open it, and fetch the qualifying tuples sequentially. Moreover, one can specify that a query or collection of queries is to be a transaction [ESWA76, GRAY78]. The DBMS provides serializability and an atomic commit for such transactions.

There are several drawbacks to such an interface when used to implement a browser such as the one discussed above. First, the relation editor can scroll

[1] A database cursor is an embedded query language concept not the cursor displayed on a CRT.

backwards, thereby requiring that the cursor be repositioned to a previously fetched tuple. This feature is not supported by a conventional programming language interface (PLI). Second, current PLI's return one record at a time. When the user scrolls forward or backward, a browsing program would prefer that the DBMS return as many records as will fit on the screen. The program issues one request and receives several records. This protocol simplifies the browsing program code.

Next, the browser must scan forward or backward to the first tuple that satisfies a predicate. This function is needed to implement the *find* operation described above. Of course, the predicate could be tested in the application program but would duplicate function already present in the DBMS. A cleaner and more efficient solution would be to use the DBMS search logic through a new programming language interface.

Lastly, to implement the *update* operation, the relation editor must be able to commit updates incrementally during the execution of a single query. Conventional transaction management facilities do not support this kind of update.

This paper describes an application program interface that supports the data manipulation and transaction management facilities required to implement database browsers. The basic idea is to have the database management system support an object, called a *portal*, that corresponds to the data returned by a single query and allow a program to retrieve data from it. Figure 2 shows a general model for the proposed system. The DBMS manages portals and allows a program to selectively retrieve or update data from the portal with a new collection of DBMS commands.

A portal can be thought of as a relational *view* that is *ordered*. The query that defines the portal retrieves the data in some particular sequence which establishes the ordering of tuples in the portal. Each tuple will have an extra

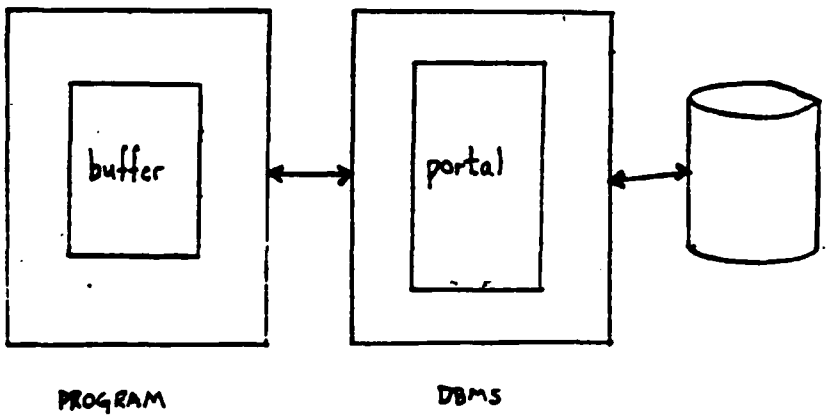


Figure 2. General Model for Portals.

field that contains a unique sequence number, called a *line identifier* (*LID*) [STON82a] that represents the position of the tuple in the portal. Line identifiers are automatically updated when tuples are inserted into or deleted from the portal so the position of each tuple is always represented by the line identifier.

Commands are provided which return collections of portal tuples to the application program. For example, a program can request tuples which:

- match a predicate (e.g., "all employees over 40"),
- scroll from the current position of the cursor (e.g., the tuples whose LID exceeds the LID of the tuple pointed at by the cursor by less than 24), or
- surround a particular tuple in the portal (e.g., the tuples with an LID within 12 of the LID of the tuple corresponding to Jones)

Changes made to the data in a portal are propagated to the relations that define it when the update is committed. Six commit modes are supported so

that different forms of concurrency control can be implemented by an application program. In addition to modes that allow one or more queries to be treated as an atomic transaction, a mode is provided that allows a transaction to be committed incrementally.

This paper describes the design and one proposed implementation of this new application program interface. Section 2 presents the design of the portal abstraction. Section 3 describes a new collection of tactics that a database system can use to implement portals. Section 4 discusses some issues in designing versions of the language constructs for different programming languages and contains some other comments on their implementation and use.

2. APPLICATION PROGRAM INTERFACE

The application program interface includes language constructs to define a portal, to open and close a portal, to fetch tuples from a portal, to update tuples in a portal, and to further restrict a portal. A portal is defined by specifying a query that selects the tuples that are in it. The general format of a portal definition is similar to the definition of a cursor [ASTR76] and is[2]

let portal **be** (target-list) [**where** qualification]

where *portal* is the name of the portal, *target-list* is a comma separated list of expressions that define the columns or attributes in the portal, and *qualification* is a predicate that determines which tuples are in the portal. For example, given an employee relation with the following attributes

EMP (name, address, age, salary, years-service, dept)

the command

let p **be** (EMP.name, EMP.salary, birthyear = 1982 – EMP.age)
where EMP.salary > 25000

[2] [x] indicates that x is optional.

defines a portal, named *p*, that contains the name, salary, and birthyear of employees whose salary is greater than $25,000.

The query that defines a portal can be a multiple variable query. For example, given a department relation

DEPT (dname, mgr, floor, budget)

a portal that contains employee and department information can be defined by

let p1 **be** (EMP.name, EMP.dept, DEPT.floor) **where** EMP.dept = DEPT.dname

This portal contains the name, department, and department floor for all employees. The portal query can also include programming language variables so that it can be defined at run-time. For example, the following declaration

let p2 **be** (EMP.name) **where** EMP.salary > x **and** q

includes two program variables, *x* and *q*, that allow the employee's salary and some other predicate (e.g., "EMP.age < 20") to be substituted at run-time.

The definition of a portal causes the query to be parsed and stored by the DBMS. Then, opening a portal causes the values of run-time variables in the portal query to be passed to the DBMS. Depending on the implementation tactic chosen by the DBMS, the query might be executed and a temporary relation created to store the portal data. Other implementation tactics are described in the next section. For now, a portal can be thought of as a view. The open command also specifies the program variable into which data will be fetched and an optional lock mode that selects a concurrency control mechanism for the portal. The general format of the open command is

open portal **into** variable [**with** lock-mode = n]

where *portal* is the name of the portal, *variable* is a program buffer, and *n* is an integer that identifies a lock-mode. The program buffer is an "array of records" declared in the application program which determines the maximum number of

tuples that can be retrieved from the portal by one command. Lock modes and transaction management are discussed below.

A portal remains open until it is explicitly closed by a close command. The format of a close command is

close portal

Figure 2 shows a PASCAL program fragment that declares a buffer, defines a portal, and opens it. The buffer, named *buf*, has a field with the same name as each attribute in the portal. Notice that even though the line identifier was not explicitly defined in the target-list of the portal definition, it is included in the buffer record. A column, named *LID*, is implicitly defined for each portal.

Data can be retrieved from the portal and stored in the program buffer by the fetch command. For example, the command

fetch buf

fetches data from *p* and stores it into *buf*. When the program run-time environ-

```
        { declare buffer }
var  buf: array [1..10] of
              record
                    LID: integer;
                    name: array [1..20] of char;
                    salary: real;
                    age: integer
              end

begin
      ...
  let p be (EMP.name, EMP.salary, EMP.age) where EMP.salary > 25000
  open p into buf
      ...
end
```

Figure 2. PASCAL program fragment that declares a portal.

ment passes this command to the DBMS, it also passes the number of records that can be stored in the buffer The DBMS returns the number of tuples requested to the program. The attribute values returned from the portal are automatically converted to the appropriate data types and stored in the buffer.

A built-in function is provided that indicates how many records were actually stored in the buffer by the last fetch command. The programmer can use this function to determine if any data was returned or if the buffer is only partially filled. For example, if the portal in figure 2 contained only 5 records, the fetch command above would not fill the buffer. On the other hand, if the portal contained 50 tuples, the command would fetch only the first 10 tuples because only that number can fit in the buffer. The program can retrieve the next 10 tuples by executing a fetch command with a **where**-clause as follows:

fetch buf **where** p.LID > 10

This command fetches 10 tuples beginning with tuple number 11. Notice that the portal name, in this case *p*, is used to reference tuples in the portal.

A fetch command can have an arbitrary qualification that will restrict the tuples retrieved to those that satisfy a predicate. For example, the program might want to retrieve employees under 20 who make more than $40,000. The command to retrieve these records is

fetch buf **where** p.age < 20 **and** p.salary > 40000

The fetch command can also be used to retrieve data by position and to search forwards or backwards. The general format of the fetch command is:[3]

fetch [**previous**] buffer
[{**where** | **after** | **before** | **around**} qualification]

A *position* fetch uses the keyword **after**, **before**, or **around** rather than **where**. A fetch with an **after**-clause indicates that the first tuple that satisfies the

[3] {x|y} indicates that x or y must appear.

qualification and the tuples immediately after it in the portal ordering are to be retrieved. For example, if the following command was executed on the portal in figure 2 it would retrieve 10 tuples beginning with tuple number 40:

fetch buf **after** p.LID = 40

Tuples 40 to 49, if they exist, would be stored in *buf*. The tuple that satisfies the qualification (i.e., tuple number 40) is stored in *buf[1]*. Subsequent returned tuples follow the selected one in *LID* order and do not necessarily satisfy the qualification. In contrast, all tuples returned by a restriction fetch (i.e., one that includes a **where**-clause) must satisfy the qualification.

The keyword **before** indicates that the first tuple that satisfies the qualification should be stored at the end of the buffer. Consequently, the buffer will contain the qualifying tuple and the tuples that immediately precede it. The keyword **around** indicates that the qualifying tuple should be stored in the middle of the buffer and the tuples immediately before and after it will be fetched.

The qualification in a position fetch can be an arbitrary predicate such as

... **after** p.LID > 10 **and** p.age < 25

which retrieves tuples beginning with the first one found after tuple number 10 that satisfies the qualification on age. This facility can be used to implement a search operation which scans for the first record after the current one that satisfies a user-specified predicate. The following command fetches the appropriate data

fetch buf **around** p.LID > n **and** q

where *n* is the *LID* of the current record and *q* is a string variable that contains the user-specified predicate. Most browsers also allow users to search backwards. The **fetch previous** command can be used to implement this function. It scans backward through the portal rather than forward. For example, the com-

mand

fetch previous buf **before** p.LID < n **and** q

searches for the first record before the current one that satisfies a search predicate.

The qualification in a fetch command can be any boolean combination of terms involving portal variables (e.g., "p.age = 40") and application program variables (e.g., "q from the example above"). It is also possible to support qualifications involving join terms to other data base relations.

A command is provided which allows a programmer to restrict the portal to a smaller subset of the data that it currently contains. The format of the restrict command is:

restrict portal **where** qualification

This command removes from the portal all tuples which do not satisfy the qualification. For example,

restrict p **where** p.age > 25

removes all employees 25 and under from the portal. A restrict command is equivalent to defining a new portal with a qualification obtained by AND'ing the new qualification to the one that defined the portal.

The portal abstraction also includes update commands to insert, delete, and replace tuples in the buffer. Appropriate commands are also passed to the DBMS which change the portal so that subsequent fetches will see the updated data. When a transaction is committed, portal changes become visible to other DBMS users.

Because portals are defined by queries, some updates cannot be unambiguously mapped onto the underlying relations. This problem is identical to the problem of updating relational views [DAYA78, STON75]. However, since portal

updates affect single tuples only, several special purpose view update algorithms appear possible for this restricted case.

The general format of the replace command is

replace buffer-reference (target-list)

where *buffer-reference* is a program reference to a record in the buffer (e.g., *buf*[*i*]). For example, the following command changes the age of the tuple stored at *buf*[*4*]:

replace buf[4] (age = 25)

This command does not change tuple number 4; it changes which ever tuple was last fetched into *buf*[*4*].

The insert command appends a tuple to the portal. The general format of this command is:

insert (target-list) **before** buffer-reference

This command inserts the tuple before the buffer array element referenced. The elements in the buffer are moved down to make room for the new data. Since the buffer is fixed size, the last record must be is removed from the buffer. The new record is assigned the *LID* of the element it is being inserted before. The *LID*'s of all records following the new element are incremented. The new tuple and its *LID* are passed to the DBMS which updates the portal.

The last update command allows tuples to be deleted. The format of this command is:

delete *buffer-reference*

The *LID* of the buffer element referenced is set to zero to indicate that it has been deleted. The *LID*'s of all records that follow it in the buffer are decremented. Then, the *LID* and the deleted record value are passed to the DBMS which updates the portal.

Update commands are passed to the DBMS which records the changes so that subsequent fetches will return the new data. The lock mode selected when the portal is opened will determine when the update is committed to the database. The following lock modes are provided.

1. The tuples returned by a fetch command are locked, and tuples locked by the previous fetch command are unlocked. Updates are committed when the next fetch command does not span the updated tuples.

2. This option is the same as number 1 except that each update is committed immediately upon a replace, delete, or append command.

3. This option is a variant on optimistic concurrency control [BARG80, KUNG81]. The browsing program does not lock a tuple until it is deleted or replaced. When a tuple in a portal is modified, the tuple(s) from the relation(s) that define the portal are locked and the portal tuple is recreated. If the portal tuple to be modified is the same as the recreated tuple, the update is committed. Otherwise, an error is returned to the program. Append commands are committed immediately. This locking mode allows a browsing application to set no long-term read locks during a session.

4. This option is the same as number 3 except that all tuples returned by the last fetch command are locked, refetched, and compared with the recreated values. The update is committed only if they all are the same. This mode is appropriate if an update is determined by data elsewhere in the scope of the current fetch command.

5. Transactions are defined explicitly by the program. A begin and end transaction command are executed to delimit the beginning and end of the transaction. A transaction can be an arbitrary collection of fetch, insert, delete, and replace commands.

6. All commands between opening and closing a portal are considered one transaction.

The conventional definition of a transaction is that it is a collection of reads and writes which are atomically committed and serializable [GRAY78, ESWA76]. Lock modes 3-6 obey this model. For example, lock mode 4 can be implemented as follows:

```
begin transaction
        recreate the most recently fetched tuples
        if tuples changed
                then abort the replace or delete
                else update relation(s)
end transaction
```

Lock modes 1 and 2, on the other hand, do not correspond to any atomically committed and serializable collection of reads and writes. They both require that locks be held after the end of an atomically committed action.

The next section describes several tactics for implementing portals.

3. IMPLEMENTATION STRATEGIES

This section describes four strategies for implementing the portal abstraction. It is expected that a data manager would implement most (or all) of them. For each portal the DBMS would select one based on the estimated size of the portal and hints from the user program. Selecting an implementation for a portal is analogous to optimizing a query in a conventional relational system. This section also describes the transaction management facilities needed to implement the six lock modes for portals.

3.1. Portal Implementation

The first strategy for implementing portals is to create an ordered temporary relation that contains the portal data. Portal commands would then be translated into conventional queries on this temporary relation. A tuple in the

temporary relation must contain a column for each attribute in the portal and a disk pointer[4] to each tuple used to construct it. For example, given the portal

let p **be** (*EMP.name*, EMP.age, EMP.dept, DEPT.mgr)
where EMP.dept = DEPT.dname

defined on the *EMP* and *DEPT* relations described in section 2, a temporary relation is created for this portal by executing the following query

retrieve into TEMP(EMP.name, EMP.age, EMP.dept, DEPT.mgr,
EMP_TID=EMP.TID, DEPT_TID=DEPT.TID)
where EMP.dept = DEPT.dname

If *TEMP* is organized as an ordered relation [STON82a], the DBMS will automatically create and maintain the *LID* attribute using an auxiliary storage structure called an *ordered B–tree* (*OB–tree*). An OB–tree is similar to a B^+–tree (i.e., data is stored in the leaves of the tree and a multi-level index is provided to access the data as indicated in figure 3). The leaf pages in the tree contain pointers to the tuples in the relation (i.e., *TID*'s). The LID ordering of the tuples is represented by the order of the *TID*'s in the leaf pages. Hence, traversing the leaf pages from left to right scans the tuples in *LID* order (i.e., the first *TID* in the leftmost page is the tuple with *LID* 1). Non-leaf pages contain a pointer to the next level of the index or a leaf page and a count of the number of tuples in that subtree.

The tree structure and the tuple counts can be effectively used by the DBMS to retrieve or update tuples based on their *LID*. For example, to find the i-th tuple, the DBMS begins at the root page and selects the subtree that contains the tuple by performing a simple calculation. Assuming that s_i is the number of tuples in the first i subtrees, i.e.,

$$s_i = \sum_{j=1}^{i} count_j$$

[4] In a relational DBMS, a pointer to a tuple in a relation is called a *tuple identifier* (*TID*).

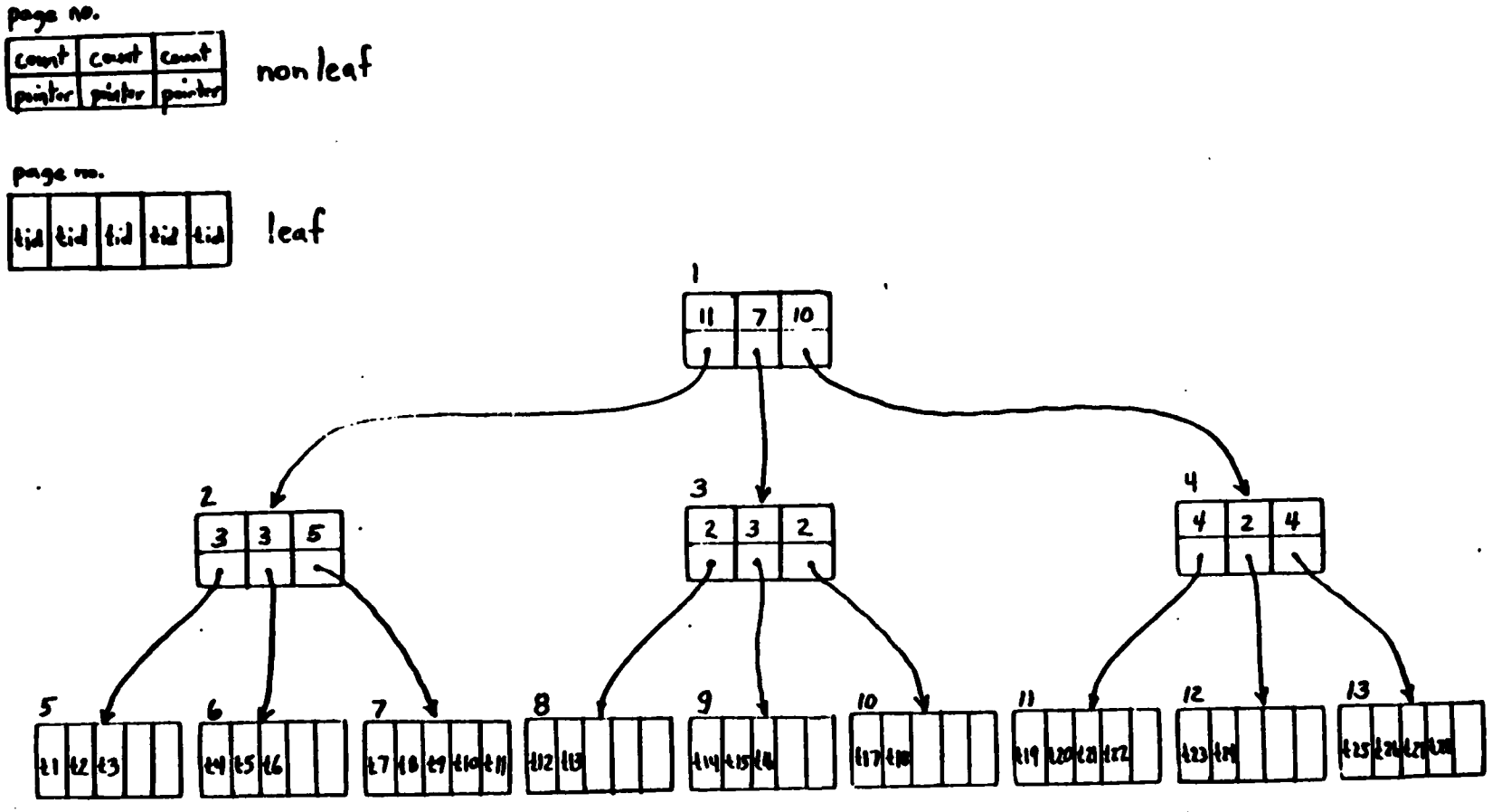


Figure 3. An OB–tree.

the subtree that contains the l-th tuple is pointed to by the entry at

$$\min_i \{ s_{i-1} < l \leq s_i \}$$

This process is performed iteratively until the algorithm reaches a leaf page which is guaranteed to contain the tuple. The calculation at intermediate levels of the tree to select a subtree must take into account the number of tuples that precede the first tuple in the subtree. Assuming that this number is x, the calculation to select the correct subtree for intermediate levels is

$$\min_i \{ x+s_{i-1} < l \leq x+s_i \}$$

The value for x is s_{i-1} at the next outer level. The *TID* for the l-th tuple is stored in the leaf page at entry $l - x$.

For example, in figure 3 to find the tuple with LID 17, the algorithm will examine page 1 and select the second subtree because 17 is between 11 (s_1) and 18 (s_2). Examining page 3 with x equal to 11, the algorithm selects page 10 because 17 is between 16 ($x + s_2$) and 18 ($x + s_3$). Page 10 is a leaf and the *TID* for tuple 17 is stored in the first entry ($l - x$).

Insertions into an OB–tree are implemented by inserting a TID for a new tuple into the appropriate leaf page and updating the counts. A standard B–tree split algorithm is used if the leaf page is full [KNUT73]. Deletions and replaces are implemented in a similar way. A complete description of these operations and a prototype implementation of OB–trees are described in [LYNN82].

In the first implementation strategy, the DBMS executes portal commands by transforming them into queries on the temporary relation. For example, the fetch command

fetch buf **where** p.age < 25

is implemented by executing the query

retrieve (TEMP.LID, TEMP.all) **where** TEMP.age < 25

Recall that the number of records that can fit in the program buffer is passed to the DBMS along with the command so that only the requested number of tuples are returned.

A position fetch is implemented by executing two retrievals. Suppose the position fetch was

fetch buf **after** p.LID > 10 **and** p.age < 25

and that the program buffer can hold n records. First, the following query is executed to find the LID of the first qualifying tuple

retrieve (l = min(TEMP.LID)) **where** TEMP.LID > 10 **and** TEMP.age < 25

Then the DBMS can execute a query to return n tuples beginning with the l-th tuple. The query to retrieve these tuples is

retrieve (TEMP.LID, TEMP.all) **where** l ≤ TEMP.LID **and** TEMP.LID ≤ l+n-1

After and **around** position fetches can be implemented using a similar technique.

Fetch previous commands can be implemented by scanning the OB–tree backwards. Fetch commands that include joins with other relations are easy to implement because the portal is stored as a relation. Update commands on the portal are implemented by executing queries to update the temporary relation and writing an intentions list that will be used by the transaction manager to update the primary relation(s). Finally, restriction commands are implemented by creating a new temporary.

The advantages of this implementation are that large portals can be browsed and that forward and backward searching can be implemented efficiently. The disadvantages are the time and space it takes to create the temporary relation.

A possible improvement to this strategy is to create the temporary relation incrementally. At any time the temporary relation contains all tuples with *LID*'s less than the maximum *LID* that has been fetched thus far. If the data required by a fetch command is in the temporary relation, a retrieval is executed to fetch it. Otherwise, the portal query is resumed to retrieve more data into the temporary and the retrieval is executed. An update command can only modify data that has already been fetched so the data to be changed must be in the temporary.

Incrementally constructing the temporary reduces the time needed to open the portal because the retrieval to create the temporary is deferred. However, this implementation introduces more variability in the time to execute a fetch command because the portal query may have to be resumed. The space required for the temporary will be reduced if the user specifies a query that generates a large portal, but does not examine all of the data in it.

Another improvement is possible when the relation on which the portal is defined is already maintained by the DBMS as an ordered relation. If the portal definition selects all fields from this relation with no restriction, then the DBMS can directly utilize the underlying primary relation structure and no copy is required.

The second strategy for implementing portals is to store the temporary relation in primary memory. The representation in memory can use an OB—tree or a conventional data structure, such as an AVL-tree, hash table, or array. The implementation of portal commands is identical to that described above. The advantage of this implementation is that portal commands will be faster because primary memory is faster than secondary storage. Update commands will also be faster because only the intentions list has to be written to disk. The disadvantage of this implementation is that only small portals can be stored in pri-

mary memory. Of course, a main memory implementation can also be incrementally materialized to reduce space requirements.

The third strategy for implementing portals is to store pointers to the tuples in the primary relations in the temporary relation (i.e., the temporary is a kind of secondary index). For example, given the portal definition

let p **be** (EMP.all) **where** EMP.salary > 20000

the DBMS does not have to make a copy of the data in the *EMP* relation. The ordered temporary relation could be defined by

retrieve into TEMP(EMP.TID) **where** EMP.salary > 20000

Fetch commands that involve only the *LID* attribute can be implemented by restricting *TEMP* to the qualifying entries and using the *TID*'s to access the *EMP* tuples. The advantage of this implementation is that it reduces the space required to store the temporary relation. The disadvantage is that it requires an extra disk read to fetch the data so portal commands will be slower.

The fourth strategy for implementing portals is to materialize the portal dynamically and to buffer only the amount of data needed by the current fetch command. For example, suppose the browsing program issued a sequence of fetch commands that scrolled forwards through the portal. The DBMS would execute the portal query to generate tuples to be returned by the current command and would keep them in main memory buffers. The next fetch command would be implemented by continuing the portal query and discarding the tuples buffered for the previous fetch. If the browsing program issues a fetch command that requires data that has already been discarded, the portal query must be restarted at the beginning.

The advantage of this implementation is that very large portals can be browsed without having to make a copy of the data. The disadvantages are that some commands will be slow and that fetch previous commands cannot be

implemented efficiently. An obvious improvement to this strategy is to buffer more data than was returned by the last command which would allow some fetch previous commands to be implemented.

3.2. Concurrency Control

The implementation of the six lock modes for portals can use a conventional transaction manager that locks physical entities and supports operations to begin, commit, and abort transactions. The general strategy is to update the temporary relation when the update command is executed. In addition, updates for the primary relation(s) are generated and written to a log. These updates are either committed immediately (lock mode 2) or at a later time (lock modes 1 or 3-6).

Lock modes 1 and 2 can be used only if the portal is implemented by dynamic materialization (i.e., strategy four discussed above). An update is committed when the tuple is not included in the next fetch command (i.e., it is removed from the buffers). The DBMS locks tuples which are buffered in main memory. Locks can be released immediately if the portal is defined on a single primary relation. If a portal is defined by a join, the lock is released only if the tuple is not used to construct another portal tuple which is currently locked. For example, suppose the portal definition was

```
let p be (EMP.name, EMP.dept, DEPT.floor, DEPT.mgr)
        where EMP.dept = DEPT.dname
```

and two employees, say Smith and Jones from the toy department, are in the DBMS buffer. Consequently, the two *EMP* relation tuples and the *DEPT* relation tuple would be locked. If Smith's tuple was removed from the portal, the lock on his tuple in the *EMP* relation can be released. However, the lock on the toy department tuple could not be released because it is used to construct Jones' tuple in the portal. In other words, the buffer must be searched to see if the

department tuple is used elsewhere before that lock can be released.

Locks do not have to be released on every fetch. For example, it may be advantageous to perform lock releases periodically. Releasing locks is analogous to garbage collection of free space by a programming language run-time system. However, in contrast to garbage collection which is performed when free space is exhausted, a DBMS wants to release locks as soon as possible to increase parallelism.

Lock mode 2 differs from lock mode 1 only in the time at which updates are committed back to the underlying primary relation(s). Locking is implemented the same way it is for lock mode 1.

Lock mode 3 which requires refetching the tuple being changed can be implemented as follows. The primary relation(s) are not locked. When a replace or delete command is executed, the *TID's in the temporary relation* are used to lock and refetch the values from the primary relation(s). The update is aborted if the value in the primary relation is different than the value in the temporary relation. Otherwise, the primary relations are updated and the locks are released. Lock mode 4 can be implemented in the same way.

Lock mode 5 and lock mode 6 can be implemented in an obvious way. In lock mode 5, the program indicates when the begin and commit operations should be executed. In lock mode 6, the DBMS begins the transaction when the portal is opened and commits updates when the portal is closed.

4. DISCUSSION

This section discusses several issues concerning the design and implementation of the portal abstraction. First, the language constructs presented in section 2 map a portal into a buffer which is a static 1-dimensional array. The constructs can be generalized to dynamic and n-dimensional arrays. If the pro-

gramming language into which the constructs are embedded has dynamic arrays, the size of the program buffer can be redefined at run-time. The DBMS can pass a count of the number of records that will be returned by a fetch command before the records are returned. The run-time support routines in the user program can dynamically allocate an array to hold the returned records. This would relieve the program of executing multiple fetch commands when the number of returned tuples exceeded the static buffer size.

Ordered relations can also be generalized to n dimensions [STON82a]. In this case a relation can have several LIDs, one for each dimension. The language constructs discussed in section 2 can be easily generalized to support a portal with multiple LIDs which is mapped to an n-dimensional buffer. This feature would be especially valuable to browsers such as SDMS [HERO80] which implement 2 dimensional scrolling.

The second design issue concerns how the portal commands are integrated into existing query language embeddings that do not have an explicit open command (e.g., EQUEL [ALLM76]). The basic idea is to generalize the notion of a range variable to include portal constructs. For example, the command

```
range of buf is p(EMP.all)
   where EMP.age < 40
   with lock-mode=3
```

would be equivalent to

```
let p be (EMP.all) where EMP.age < 40
open p into buf with lock-mode=3
```

Lastly, a database system that implements portals must be able to save and restore the currently executing query because programs can open multiple portals and because several implementation strategies discussed in Section 3 are based on restarting the portal query.

5. CONCLUSIONS

A new application program interface to a relational database system has been described which makes it easier to implement database browsers. The interface is based on the concept of a *portal* that supports querying and updating an ordered view. Several lock modes were suggested that can be used to implement browsing transactions with varying consistency and parallelism requirements.

Acknowledgements

Several people have contributed ideas that have been incorporated into this proposal. We want to thank Paul Butterworth, Joe Kalash, Richard Probst, Beth Rabb, and Kurt Shoens for their contributions.

References

[ALLM76] Allman, E. at. al., "Embedding a Relational Data Sublanguage in a General Purpose Programming Language," Proc. ACM-SIGPLAN-SIGMOD Conference on Data Abstraction, Definition and Structure, Salt Lake City, Utah, March 1976.

[ASTR76] Astrahan, M. M., et. al., "System R: A Relational Approach to Data," ACM TODS, June 1976.

[BARG80] Bhargava, B., "An Optimistic Concurrency Control Algorithm and Its Performance Evaluation Against Locking," Proc. International Computer Symposium, Taipai, Taiwan, Dec. 1980.

[CATE80] Catell, R., "An Entity-based Database User Interface," Proc. 1980 ACM-SIGMOD Conference on Management of Data, Santa Monica, CA, May 1980.

[DAYA78] Dayal, U., and Bernstein, P., "On the Updatability of Relational Views," Proc. 4th Very Large Data Base Conference Montreal, Canada, October 1978.

[ESWA76] Eswaren, K., et. al., "On the Notion of Consistency and Predicate Locks in a Relational Database System," CACM, November 1976.

[GRAY78] Gray, J., "Notes on Data Base Operating Systems," IBM Research, San Jose, CA, Report RJ 2188, February 1978.

[HERO80] Herot, C., "SDMS: A Spatial Data Base System," TODS, December 1980.

[JOY79] Joy, W., "The vi Text Editor," unpublished working paper.

[KNUT73] Knuth, D., "The Art of Computer Programming, Vol 3: Sorting and Searching," Addison Wesley, Reading, Mass., 1973.

[KUNG81] Kung, H. and Robinson, J., "On Optimistic Methods for Concurrency Control," ACM TODS, June 1981.

[LYNN82] Lynn, N., "Implementation of Ordered Relations in a Data Base System," University of California, Berkeley, CA, Masters Report, Sept. 1982.

[MARY80] Maryanski, F., "Query By Forms," (unpublished presentation).

[ROWE79] Rowe, L. and Shoens, K., "Data Abstraction, Views and Updates in RIGEL," Proc. 1979 ACM-SIGMOD Conference on Management of Data, Boston, Mass. May 1979.

[ROWE82] Rowe, L. and Shoens, K., "FADS - A Forms Application Development System," Proc. 1982 ACM-SIGMOD Conference on Management of Data, Orlando, FL, June 1982.

[SCHM77] Schmidt, J., "Some High level Language Constructs for Data of Type Relation," ACM TODS, Sept. 1977.

[STAL81] Stallman, R.M., "EMACS The Extensible, Customizable Self-Documenting Display Editor," Proc. 1981 ACM-SIGPLAN/SIGOA Symp. on Text Manipulation, SIGPLAN Notices, 16, 6, June 1981.

[STON75] Stonebraker, M., "Integrity Constraints and Views by Query Modification," Proc. 1975 ACM-SIGMOD Workshop on Management of Data, San Jose, CA, May 1975.

[STON82] Stonebraker, M. and Kalash, J., "TIMBER: A Sophisticated Relation Browser," Proc. 8th International Conference on Very Large Data Bases, Mexico City, Mexico, September 1982.

[STON82a] Stonebraker, M., et. al., "Support for Document Processing in a Relational Database System," Electronics Research Laboratory, Memo M82/15., March 1982.

[WASS79] Wasserman, A., "The Data Management Facilities of PLAIN," Proc. 1979 ACM-SIGMOD Conference on Management of Data, Boston, Mass., May 1979.

[ZLOO82] Zloof, M., "Office-by-Example: A Business Language That Unifies Data and Word Processing and Electronic Mail," IBM Systems Journal, Fall 1982.

An Implementation of Hypothetical Relations

by

John Woodfill and Michael Stonebraker

DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

UNIVERSITY OF CALIFORNIA

BERKELEY, CA.

ABSTRACT

In this paper we develop a different approach to implementing hypothetical relations than those previously proposed. Our design, which borrows ideas from tactics based on views and differential files, offers several advantages over other schemes. An actual implementation is described and performance statistics are presented.

1. INTRODUCTION

The motivation for, and applications of hypothetical relations (HR's) were introduced in [STON80]. They can be used to support "what if" changes to a data base and offer a mechanism for debugging applications programs on live data without fear of corrupting the data base. The suggested implementation in [STON80] involved a differential file [SEVR76]. In [STON81], supporting HR's as views [STON75] of the form W = (R UNION S) - T was suggested. In this case an implementation only requires extending a relational DBMS and its associated view mechanism with the UNION and - operators. Moreover, R can be a read-only relation while S and T are append only. As a result, hypothetical relations may

offer cheap support for crash recovery and logging. Unfortunately, there are problems with treating HR's as views. We first examine these problems and show general solutions in Section 2. Next we combine these solutions in Section 3 into a new mechanism for supporting HR's. Our proposal has several similarities but a different orientation from one in [KATZ82]. We then describe our implementation in Section 4. Finally we analyze the performance of this implementation in Section 5.

2. PROBLEMS AND SOLUTIONS

Proposals for hypothetical relations as views contain various flaws which must be removed before a realistic implementation can be attempted.

2.1. A Known Problem

[STON81] points out that the implementation of hypothetical relations as W = (R UNION S) - T is flawed in the case where one wants to re-append a tuple which has been deleted, as shown by the example in figure 1. Initially there is a tuple in relation R corresponding to Eric. Following the algorithm in [STON81], the tuple can be deleted by inserting it into relation T. Lastly a user re-appends Eric and an appropriate tuple is inserted into S. Unfortunately, the resulting relation, W does not contain the re-appended tuple, since (R UNION S) is the same as R, and R - T is empty.

2.2. A Solution

As noted in [Agra82], this problem can be solved by adding a timestamp field to the relations S and T, and modifying the semantics of the DIFFERENCE operator, "-". There will be no timestamps for the relation

R

name	salary
eric	10000

S

eric	10000

T

eric	10000

Figure 1.

R; hence these tuples can be thought of as having a timestamp of zero.

The timestamp field is filled in with the current time (from a system clock, or any other monotonicaly increasing source of timestamps) whenever a tuple is appended to S or T. For any relations A and B with timestamps as described, the DIFFERENCE, A - B is defined as all tuples a in A for which there is no tuple b in B such that

(1) DATA(a) = DATA(b)

and

(2) TIMESTAMP(a) < TIMESTAMP(b)

The definition of R UNION S is unchanged, except for the addition of a timestamp field in the result which contains either the timestamp of a tuple in S, or a zero timestamp for a tuple in R. If tuples with identical DATA appear in both R and S, the newer timestamp (from S) is chosen for the result tuple.

In the above example, the timestamp of Eric's tuple in T would be newer than that of Eric's tuple in R (zero), but would be older than the timestamp of Eric's tuple in S; hence, (R UNION S) - T would be

equivalent to S, and W would contain the re-appended tuple.

2.3. A New Problem

The addition of timestamps solves the problem of appending deleted tuples. However, this solution is not free from problems. Consider the case of a second level hypothetical relation, W' = (W UNION S') - T', as shown in figure 2. Suppose Eric was given a 20 percent raise in W' at timestamp 10 which caused the indicated entries in S' and T'. Since no updates have occurred in W, S and T are empty. Now suppose a user gives Eric a 50 percent raise in W at timestamp 20, which results in the entries for S and T shown in figure 3. According to the algorithm above, W' would contain two tuples for Eric, one with salary 15,000, and one with salary 12,000. The problem is that the tuple in T' no longer functions to exclude Eric from W UNION S' and hence an unwanted Eric tuple is present.

There are at least two choices for the proper semantics for W' under this update pattern:

R

name	salary
eric	10000

S

name	salary	t-stamp

T

name	salary	t-stamp

S'

name	salary	t-stamp
eric	12000	10

T'

name	salary	t-stamp
eric	10000	10

Figure 2, Eric's 20% raise in W'.

R

name	salary
eric	10000

S

name	salary	t-stamp
eric	15000	20

T

name	salary	t-stamp
eric	10000	20

S'

name	salary	t-stamp
eric	12000	10

T'

name	salary	t-stamp
eric	10000	10

Figure 3, Eric's 50% raise in W.

1) Eric's salary is set to the latest value, in this case the 15,000 from W.

2) Eric's salary is set to 12,000, corresponding to the original update of W'.

We choose to follow the latter choice, and specify the following semantics:

> Once a tuple has been changed at level N, changes at levels < N cannot affect tuples at levels >= N.

2.4. A New Solution

These semantics can be guaranteed by the addition of a tuple identifier, and modification of the DIFFERENCE operator. A tuple identifier, TNAME, must be given to each tuple in R. Each tuple inserted into W (and thereby added to S) must also be given an identifier. Then, any inserts to S or T, which are used to replace or delete a tuple in W, must be marked with the identifier for the original tuple in R or S which they replace or delete. For any relations A and B with timestamps

and TNAMES as described, the DIFFERENCE, A - B is defined as all the tuples a in A for which there is no tuple b in B such that

(1) TNAME(a) = TNAME(b)

and

(2) TIMESTAMP(a) < TIMESTAMP(b)

To guarantee that our chosen update semantics hold, tuples in A - B are given timestamps of zero. Hence, at a second level, each tuple in S' and T' will have a newer timestamp than its corresponding tuple in W.

In our example the identifier of all of the five Eric tuples from figure 3 will be identical. Since the timestamp of the tuple in W is treated as being older than that of the tuple in T', only Eric's tuple from S' will be contained in W'.

A similar method is proposed in [KATZ82], to solve this problem.

3. A MECHANISM

Given these modifications to S, T and the DIFFERENCE operator, an HR of the form W = (R UNION S) - T no longer has its original conceptual simplicity. Moreover, support for HR's becomes considerably more complex than simply implementing UNION and - as valid operators in a DBMS. Consequently, we have designed a mechanism based on differential file techniques which builds on the above developments. The goal is to provide a single-pass algorithm with proper semantics that will support arbitrary cascading of HR's. The next two sections describe our data structure and algorithm in detail.

3.1. The Differential Relation

Each hypothetical relation W, built on top of a real or hypothetical relation B, has an associated differential file D, which contains all columns from B plus plus five additional fields. For example, the differential relation D for the base relation R from Section 2 is shown in figure 4. "Name" and "salary" are the attributes from R. The fields "mindate" and "maxdate" are both timestamps. "Mindate" is exactly the timestamp as defined above, while "maxdate" is another timestamp to be explained in section 4.2. The fields "level" and "tupnum" are used to identify the tuple which this tuple replaces or causes deletion of. Each hypothetical relation is assigned a level number as indicated in figure 5. All real relations are at level zero, and an HR built from a real relation is assigned a level of one. Then an HR built on top of a level one HR is given a level of two. Here the column "level" identifies the level number of a particular tuple, while the column "tupnum" is a unique identifier at that level. Together "tupnum" and "level" comprise the unique identifier, TNAME, of a tuple. Values for "tupnum" are just a sequentially allocated integers. The last field in D, "type," marks what form of update the tuple represents; thus, it has three values, APPEND, REPLACE, and DELETE.

The following examples will illustrate the use of these extra

name	c12
salary	i4
mindate	i4
maxdate	i4
tupnum	i4
level	i1
type	i1

Figure 4.

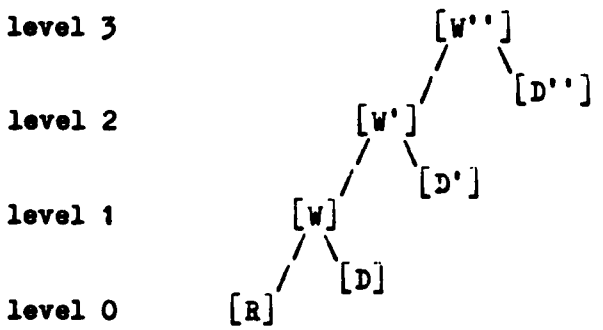


Figure 5.

fields. A precise algorithm is presented in Section 3.2.

Suppose the relation R has the following data:

name	salary
fred	4000
sally	6000

tupnum of this tuple is 0
tupnum of this tuple is 1

Figure 6.

Initially W is identical to R, and D is empty.

Running the following QUEL command:

append to W (name = "nancy", salary = 5000)

would cause a single tuple to be inserted into D as follows:

name	salary	mindate	maxdate	tupnum	level	type
nancy	5000	30	**	0	1	APPEND

Figure 7.

The 30 stored in "mindate" is simply the current timestamp, and the "type" is clearly APPEND. Since there is no corresponding tuple at level 0, which the tuple replaces, the fields "level" and "tupnum" are

set to identify the tuple itself (i.e. "level" = 1, "tupnum" = 0)

Suppose we now change the salary of Sally as follows:

```
range of w is W
replace w (salary = 8000) where w.name = "sally"
```

After this update, D looks like:

name	salary	mindate	maxdate	tupnum	level	type
nancy	5000	30	**	0	1	APPEND
sally	8000	40	**	1	0	REPLACE

Figure 8.

"Mindate" is 40, the current timestamp. The tuple which we are replacing in R has an identifier of (level = 0, tupnum = 1) (see figure 6).

Suppose we delete the tuple just replaced:

```
delete w where w.name = "sally"
```

The resulting form of D is:

name	salary	mindate	maxdate	tupnum	level	type
nancy	5000	30	**	0	1	APPEND
sally	8000	40	**	1	0	REPLACE
		50	**	1	0	DELETE

Figure 9.

Since this operation is a delete and "name" and "salary" are no longer important, they are set to null. "Tupnum" and "level" are the same as in in figure 8, since they refer to the same tuple.

Suppose we now replace the tuple appended above; eg:

```
replace w (name = "billy") where w.name = "nancy"
```

The resulting form of D is:

name	salary	mindate	maxdate	tupnum	level	type
nancy	5000	30	**	0	1	APPEND
sally	8000	40	**	1	0	REPLACE
	0	50	**	1	0	DELETE
billy	5000	60	**	0	1	REPLACE

Figure 10.

"Tupnum" and "level" identify the original "nancy" tuple (see figure 7 above). At this point, R is unchanged, and W looks like:

name	salary	
fred	4000	unchanged
billy	5000	billy replacing nancy

3.2. The Algorithm

There are two parts to the algorithm for supporting hypothetical relations: accessing an HR, and updating an HR.

3.2.1. Accessing Hypothetical Relations

The algorithm for deriving a level N hypothetical relation W from a base relation R and a collection of differential relations D1, ..., DN is a one pass algorithm which starts with the highest level differential relation and proceeds by examining each tuple, passing through each lower level, and finally passing through the level 0 base relation. Figure 11 shows this processing order more clearly. MaxLevel is the level N of the relation H.

An auxiliary data structure, which will be called "seen-ids," is maintained during the execution of this algorithm. This data structure has one associated update routine, "see(level, tupnum)", and a boolean retrieval function, "seen(level, tupnum)". The routine see(level,

```
FOR physlevel := MaxLevel DOWN TO 0 DO
BEGIN
    WHILE (there are tuples at level physlevel) DO
    BEGIN
        tuple := get-next-tuple(physlevel);

        examine-and-process-tuple(tuple, physlevel);
    END
END.
```

Figure 11.

tupnum) inserts a TNAME into the data structure if it has not been seen before, while seen(level, tupnum) returns the value TRUE if <level, tupnum> is in seen-ids, FALSE otherwise.

The examine-and-process-tuple routine takes one or both of the following actions: it can "accept" the tuple for inclusion in H and it can call the routine "see" to place the identifier in "seen-ids". The choice of actions is dictated by Table 1.

	level0	newest	seen	type	action accept	samelevel	action see
1	yes	------	yes	-------	no	---------	no
2	yes	------	no	-------	yes	---------	no
3	no	no	------	-------	no	---------	no
4	no	yes	yes	-------	no	---------	no
5	no	yes	no	DELETE	no	yes	no
6	no	yes	no	REPLACE	yes	yes	no
7	no	yes	no	APPEND	yes	yes	no
8	no	yes	no	DELETE	no	no	yes
9	no	yes	no	REPLACE	yes	no	yes

Table 1, Processing criteria for HR's.

In applying table 1, to a particular tuple t, "level0" is a boolean condition which is "yes" if physlevel from figure 11, is zero, "no" otherwise. A tuple t at physlevel N is "newest" if (as in Section 2.4) there is no tuple tb at level N such that

(1) (t.level = tb.level and t.tupnum = tb.tupnum)
and
(2) ta.mindate < tb.mindate.

A tuple t has been "seen" when the pair <t.level, t.tupnum> has already been entered into "seen-ids". Past tests for "newest" and "seen" are presented in Sections 4.2 and 4.3. The "type" of tuple t is t.type. "Samelevel" is a boolean field to indicate if physlevel is the same as t.level. The examining and processing of a tuple is shown in figure 12.

To demonstrate this processing we will generate W from D and R in figures 6 - 10. The starting configuration is shown in figure 13. Processing starts with MaxLevel = 1 and physlevel = 1 in the differential relation D; hence, for all of this level, level0 will be false. Tuple (1) is not "newest", since tuple (4) has the same identifier, and a higher mindate. Since level0 is false, the tuple corresponds to line (3) of table 1, and the tuple is neither "accepted" nor "seen."

Tuple (2) is not "newest" either, because tuple (3) has the same identifier, and a higher mindate, and so it also corresponds to line (3) of table 1, and is neither "accepted" nor "seen."

Tuple (3) is "newest," because the only other tuple at this physlevel with the same identifier, tuple (2) has a smaller mindate. It has not been "seen," since seen-ids is empty and type is DELETE. We now determine "samelevel" by comparing the level field with physlevel. Both are 1, so "samelevel" is true and line (5) is applied. Hence, the tuple is neither "accepted" nor "seen".

Tuple (4) is also "newest," has not been "seen," and type is REPLACE. Comparing level and physlevel, we find "samelevel" is false, since the level field is 0, and physlevel is still 1. hence, (9) is the

```
examine-and-process-tuple(t, physlevel)
BEGIN
    level0     : BOOLEAN;
    newest     : BOOLEAN;
    seen       : BOOLEAN;
    type       : (APPEND, REPLACE, DELETE);
    samelevel  : BOOLEAN;

    level0 := (physlevel = 0);

    IF level0 then
    BEGIN
        newest := NULL;

        seen := seen(t.level, t.tupnum);

        type := NULL;

        samelevel := TRUE;
    END ELSE
    BEGIN
        newest := is_newest(t.mindate, t.level, t.tupnum);

        seen := seen(t.level, t.tupnum);

        type := t.type;

        samelevel := (t.level = physlevel);
    END;

    IF table-accept(level0, newest, seen, type) THEN
        accept-tuple(t);

    IF table-see(level0, newest, seen, type, samelevel) THEN
        see(t.level, t.tupnum);
END;
```

Figure 12, processing a tuple.

D

	name	salary	mindate	maxdate	tupnum	level	type
1	nancy	5000	30	**	0	1	APPEND
2	sally	8000	40	**	1	0	REPLACE
3		0	50	**	1	0	DELETE
4	billy	5000	60	**	0	1	REPLACE

R

	name	salary	
5	fred	4000	tupnum of this tuple is 0
6	sally	6000	tupnum of this tuple is 1

seen-ids = {}

Tuples "accepted"

name	salary

Figure 13, Initial structures for processing W.

correct line in table 1, and the tuple is both "seen" and "accepted". At this point, W and seen-ids look like:

name	salary
billy	5000

seen-ids = {<0, 1>}

Physlevel now changes to 0, "level0" becomes true, and we start to scan the base relation. Only lines (1) and (2) of table 1 are relevant differing in the value of "seen". To check whether a tuple has been "seen," at level 0, we look for the pair <level, location> in seen-ids. For tuple (5) this pair is <0, 0> (see figure 6) which is not in seen-ids. Hence, line (2) of table 1 is applied and we "accept" the tuple. The pair <level, location> for tuple (6) is <0, 1>, which is in seen-ids. The corresponding line is (1), so the tuple is not "accepted," and

is not "seen." We have reached the end of our scan, and have generated the relation W as follows.

name	salary
billy	5000
fred	4000

3.2.2. Updating Hypothetical Relations

All updates to an HR of level N require appending tuples to the differential relation DN at level N. The contents of the different fields in the appended tuple are specified as follows:

(A) For APPENDS and REPLACES, The data columns of DN, are filled with new data. For DELETES, the fields are NULL.

(B) Mindate, is assigned the current timestamp. (Maxdate is discussed in Section 4.2.)

(C) For APPENDS, tupnum and level are set to self-identify the inserted tuple. For DELETEs and REPLACEs tupnum and level identify the target tuple being deleted or replaced.

(D) Type is the type of the update, APPEND, DELETE or REPLACE.

4. IMPLEMENTATION

An implementation of HR's was done within the INGRES DBMS [STON76]. In order to create an HR, the following addition to QUEL was made:

DEFINE HYPREL newrel ON baserel

Once an HR has been defined, it can be updated and accessed just like an ordinary relation. Since, "baserel" can be either a regular relation, or an HR, an unlimited number of levels is allowed.

4.1. Modifications

Within the INGRES access methods, a relation is accessed first by a call to "find" which sets the range for a scan of tuples, and then "get" is called repeatedly to access each tuple in this range. It is within "get" that most of the HR algorithm is implemented. "Get" returns tuples from each differential relation, and finally the tuples from the base relation. The routines which perform REPLACES, DELETES, and APPENDs are also modified to initialize and append the appropriate tuples to the differential relation.

4.2. Newest

If tuples were appended to a differential relation at one end, and the relation were scanned from the other direction, it would be possible to tell when a tuple was the "newest" for a particular identifier by the fact that it was the first one encountered. Unfortunately, INGRES appends tuples and scans relations in the same direction. In order to be able to tell from a single pass whether a tuple is "newest", an additional timestamp field "maxdate" was added. When a tuple is appended, maxdate is set to infinity. When the tuple is REPLACED or DELETED at the same level, maxdate is updated. Thus a tuple is the "newest" if the time of the current scan is between mindate and maxdate.

4.3. Seen-ids

The data structure, seen-ids is stored in a series of main memory bit-maps, one for each level. Thus to see a tuple with tupnum Y at level L, bit Y in bitmap L is set. The boolean function "seen(L, Y)" tests whether the corresponding bit is set.

4.4. Optimization

If the base relation is organized as either a random hash structure or an ISAM structure, then the differential relations can be given a similar structure and a sequential scan of the differential relation avoided. To accomplish this, a correspondence must be established between the pages in a differential relation and those in the base relation. If a tuple would be placed on a certain page of the base relation, then the tuple in the hypothetical relation must be placed on the corresponding page in the differential relation.

To access a tuple in such a structured HR, the scan within each relation is restricted to those pages corresponding to the key of the query. For example, suppose the relation R(name, salary) is stored hashed on name and the differential relation D is stored likewise. Then, the query

```
range of w is W
retrieve (w.all) where w.name = "billy"
```

only requires accessing the appropriate hash bucket in both R and D.

There is one complication with this performance enhancement, which stems from the fact that a REPLACE command can change the hash key, and hence the page location of a tuple in a structured relation. For example, consider the following contents of R and D:

R

hashbucket	name	salary
1	suzy	3000
2	tandy	25

D

name	salary	other

Figure 14, R and D hashed on name.

Then, suppose we do the following REPLACE:

```
range of w is W
replace w (name = "tandy") where w.name = "suzy"
```

As a result, R and D would look like

R

hashbucket	name	salary
1	suzy	3000
2	tandy	25

D

name	salary	type	*
tandy	3000	REPLACE	

Figure 15, problematic hashed replace.

and the query:

```
retrieve (w.all) where w.name = "suzy"
```

would generate the result:

name	salary
suzy	3000

Despite the fact that we changed suzy's name, she appears in the result because the algorithm indicates searching hashbucket 1 of D, where there are no tuples, then searching hashbucket 1 of R, where susy's tuple appears. This tuple in hashbucket 1 of R is "accepted", because no tuples have been "seen." Unfortunately, the algorithm never searches hashbucket 0 of D to discover the correct tuple.

This problem can be solved by the addition of a fourth type of differential tuple, FORWARD. An additional FORWARD tuple is appended in hashed and ISAM differential relations whenever a REPLACE is done which inserts a tuple in a different hashbucket (or ISAM data page) than that of the target tuple. With this correction, D of figure 15 would look like:

hashbucket	name	salary	mindate	maxdate	tupnum	level	type
1		0	100	INFINITY	0	0	FORWARD
2	tandy	3000	100	INFINITY	0	0	REPLACE

Figure 16.

The processing of the query would then start in hashbucket 1 of D in figure 16, where a FORWARD tuple would be found, and the ordered pair <0, 0> would be added to seen-ids. Next, hashbucket 1 of R would be scanned, but since <0, 0> is in seen-ids, Suzy's tuple, tuple 0 of R, would not be accepted.

4.5. Functionality

With this refinement all QUEL commands have been made operational on HRs for any INGRES storage structure. Such HR's could be used as the basis for a crash recovery scheme as suggested in [STON81] with minor modifications to the our algorithms. Moreover, "snap-shots" of the state of an HR at any point in the past can be generated by setting the scan time to a time prior to the current time. Minor changes to the QUEL syntax would allow a user to run retrieval commands against an HR as of some previous point in time.

If at any time one wanted to make the changes in an HR permanent, he can use a series of QUEL statements to update the base relation using the information in the differential relations. Alternately, a simple utility could be constructed to perform the same function.

5. PERFORMANCE MEASUREMENT AND ANALYSIS

Our performance analysis is aimed at comparing the performance of standard QUEL commands on real relations versus the same ones on HRs and

our tests were run on a single user VAX-11/780. The following four commands are used to measure update performance for a real parts relation parts500(pnum, pname, pweight, pcolor) of 5000 tuples stores as a heap. Baseparts will serve both as a real relation and an HR.

```
range of b is baseparts
range of p is parts5000

(a) append to baseparts (p.all)

(b) delete b

(c) replace b (weight = b.weight + 1000)

(d) replace b (pnum = b.pnum * 1000)
```

Table 2 indicates the results of running commands a) - c) first for a real baseparts relation of 5000 tuples stored as a heap and then for baseparts as an HR. In the latter case it consists of an empty differential relation, D and a 5000 tuple real relation, R stored as a heap. Command d) was not run in this situation because it should produce comparable results to command c) for unstructured relations. Notice that real and hypothetical relations perform comparably.

To test retrieval performance we ran query (e) for four different compositions of baseparts, including

```
range of b is baseparts
(e) retrieve (m = max(b.weight))
```

a 10 tuple real relation, a 10000 tuple real relation, a 10 tuple HR and a 10000 tuple HR. The hypothetical relations had sizes of differential

query	operation	relation-type	cputime	elapsed
(a)	append	regular	24.47 secs	32 secs
(a)	append	hypothetical	26.57 secs	36 secs
(b)	delete	regular	24.38 secs	26 secs
(b)	delete	hypothetical	19.78 secs	25 secs
(c)	replace	regular	26.03 secs	28 secs
(c)	replace	hypothetical	25.03 secs	35 secs

Table 2, updates on 5000 tuples unstructured.

query	operation	relation-type	cputime	elapsed
(a)	append	regular	74.68 secs	268 secs
(a)	append	hypothetical	64.82 secs	226 secs
(b)	delete	regular	20.15 secs	31 secs
(b)	delete	hypothetical	21.32 secs	37 secs
(c)	replace	regular	42.32 secs	47 secs
(c)	replace	hypothetical	40.97 secs	59 secs
(d)	replace	regular	91.33 secs	345 secs
(d)	replace	hypothetical	89.63 secs	422 secs

Table 3, updates on 5000 tuples, hashed on salary.

relations, D, varying from 0 to 200% of the size of the R. Tables 4 and 5 show the results of these tests.

relation type	size of D	cputime	elapsed time
regular	-	0.16 secs	1 sec
hypothetical	0%	0.20 secs	1 sec
hypothetical	50%	0.26 secs	1 sec
hypothetical	100%	0.26 secs	1 sec

Table 4, Query (e) run with 10 tuple base.

relation type	size of D	cputime	elapsed time
regular	-	11.88 secs	13 secs
hypothetical	0%	13.86 secs	15 secs
hypothetical	10%	14.40 secs	15 secs
hypothetical	25%	15.22 secs	16 secs
hypothetical	50%	16.73 secs	18 secs
hypothetical	100%	18.60 secs	21 secs
hypothetical	200%	21.58 secs	30 secs

Table 5, Query (e) run with 10000 tuple base.

Query (e) was also run against a second level HR based on a first level HR with 50% of its tuples replaced. The results of this test are in table 6.

Lastly, we ran query (f) against a baseparts relation hashed on pnum.

```
range of p is parts5000
range of h is RELATION

(f) retrieve (p.weight, h.weight) where p.pnum = h.pnum
```

In this case table 7 compares performance where RELATION is either a 5000 tuple real relation hashed on pnum, or a 5000 tuple HR hashed on

hypothetical relation level	size of D	cputime	elapsed time
1	50	16.73 secs	18 secs
2	0%	17.35 secs	18 secs
2	10%	17.73 secs	19 secs
2	25%	18.52 secs	19 secs
2	50%	18.78 secs	21 secs
2	100%	20.75 secs	24 secs

Table 6, Query (e) 10000 tuples, 2 levels.

pnum, with 50% of its tuples replaced. Parts5000 is an unstructured

Query (f)			
relation	type	cputime	elapsed
hashparts	regular	131 secs	5.85 minutes
hhashparts	hypothetical	185 secs	9.88 minutes

Table 7, hashed access results.

5000 tuple relation.

Two comments are appropriate about the numbers in Table 7. First, notice that INGRES is I/O bound in both tests and elapsed time substantially exceeds CPU time. The reasons include the particular query processing tactic chosen for this query and the fact that a substantial amount of data is printed on the output device. The second point is that joins on hypothetical relations are less than a factor of two slower than those on real relations.

Thus we can see that the performance of INGRES using hypothetical relations in many types of query is never worse than a factor of its level number and usually much better. We assume that for more complex queries involving an HR, the same general result would hold.

6. CONCLUSIONS

We have described a mechanism for supporting HR's which is shown to overcome the problems of previous proposals. We have described an implementation of HR's and provided performance data to show that performance of HR's is in general no worse than a factor of one per level of HR. Moreover, in most cases, performance is considerably better than

this.

ACKNOWLEDGEMENT

This research was supported by the Advanced Research Project Agency under contract #N00039-C-0235.

REFERENCES

[AGRA82] Agrawal, R. and DeWitt, D. J., "Updating Hypothetical Data Bases," Unpublished working paper.

[KATZ82] Katz, R. and Lehman, T., "Storage Structures for Versions and Alternatives," University of Wisconsin - Madison, Computer Sciences Technical Report #479, July 1982.

[SEVR76] Severance, D. and Lohman, G., "Differential Files: Their Application to the Maintenance of Large Databases," TODS, June 1976.

[STON75] Stonebraker, M., "Implementation of Integrity Constraints and Views by Query Modification," Proc. 1975 ACM-SIGMOD Conference on Management of Data, San Jose, Ca., June 1975.

[STON76] Stonebraker, M. et. al., "The Design and Implementation of INGRES," TODS 2, 3, September 1976.

[STON80] Stonebraker, M. and Keller, K., "Embedding Expert Knowledge and Hypothetical Data Bases Into a Data Base System," Proc. 1980 ACM-SIGMOD Conference on Management of Data, Santa Monica, Ca., May 1980

[STON81] Stonebraker, M., "Hypothetical Data Bases as Views," Proc. 1981 ACM-SIGMOD Conference on Management of Data, Ann Arbor, Mich., June 1982.

Interactive Mathematical Manipulation, Typesetting, and Graphics - a Progress Report

Gregg Foster
Richard Fateman

University of California, Berkeley
July, 1983

1. RECENT WORK

We have developed and implemented a hierachical representation for mathematical expressions that includes display position, expression dimensions, font information, the notion of super- and sub-expressions, and access to the algebraic manipulation system internal form that is being represented. This is a particular representation technique which we believe can be used in other problem domains and which will work in the context of a menu-driven window oriented user interface.

1. Strophe (pronounced "stroh-fee", a musical term for putting new lyrics to old music) is a system for the representation of a tree of expression boxes. The boxes are frames with inheritance of font and position information. This is naturally done in Lisp, and is convenient to use on graphics workstations which provide immediate feedback. Lisp forms representing algebraic expressions (sums, products, powers, matrices, etc) in internal forms are accepted and converted to box frames which can be displayed.

2. Strophe's representation of mathematical expressions is largely independent of any particular algebra system. It is currently usable by Macsyma and is being prepared for use by a new mathematical representation and manipulation system (SCARAB) being developed at Berkeley.

3. Strophe has been implemented in Lisp on Sun Workstations, but the design has been kept as machine independent as possible.

4. We explored various ways of pointing at expressions and found the mouse (with keyboard back-up) to be satisfactory. A mouse driven prototype sub-expression locator/highlighter runs on the Sun Workstations (much of this work was done by R.J. Anderson) and will be merged with Strophe when address space limits are removed. The Strophe representation makes it easy to identify expression-boxes from screen coordinates and to move up or down in the expression tree.

Summary

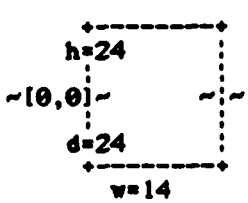
We can demonstrate high quality real-time display of mathematical expressions and the ability to access sub-expressions.

Example

a/b in Macsyma is represented internally as:

((mtimes simp) a ((mexpt simp) b -1)).

A schematic representation of one of the boxes (the outermost), as printed by a Strophe debugging tool is:




```
font: ROMAN      (normal)
  height: 12
  width:  12

out: nil
in: (b00007 b00013 b00012)

exp:
  ((mtimes simp) a ((mexpt simp) b -1))
```

The screen display is:

$$\frac{a}{b}$$

2. FUTURE WORK

Objectives

1. We wish to allow a user to preview and compose expression forms in a readable format in an interactive environment.

2. We wish to provide tools for a user of a symbolic mathematics system to point to any piece of a displayed expression and specify its alteration by a combination of mouse and keyboard commands.

3. We wish to formulate criteria for measuring the success of the SCARAB system design.

4. We wish to provide a user with more options (such as variant display formatting using tables and multi-line displays) and relevant information about his interaction with the system (such as a command history and the status of the system).

Plan

We must develop the software tools to handle the increased display-representation/mathematics-internal-representation interaction.

Requirements:

a. It must be easy to refer to an identified expression (or super-/sub expressions) in the context of manipulatory commands typed from the keyboard and indicated by the pointing device. In the case of commands which change the expression itself, these must be mapped back to the mathematics internal form and to the display.

b. Rapid redisplay with highlighting and smooth motion of selected areas must be supported.

c. Subframes to other windows should be provided.

d. There must be more and better fonts than are provided by the Sun graphics package.

e. Expression templates (a "fill in the blanks "approach) should be explored.

f. It must be easy to get to command history, process status, and other "environmental information".

Equipment

1. The Sun Workstations run Strophe and have run pieces of Macsyma and SCARAB. To run Lisp, Strophe, and an SCARAB or Macsyma together we need more address space (16 megabytes rather than 2). Sun has promised delivery of SUN-II boards with this address space in the fall.

2. A Pixel 100/AP (68000 based) UNIX system has recently been used to bring up a full Macsyma system but lacks the bitmap display capability. The performance of the 68000 is no longer an open question: it is definitely faster than the VAX 11/750 on Lisp-based applications, and with tuning and a 12megahertz chip, it may outperform a VAX 11/780.

Summary

High quality real-time composition, manipulation, and typesetting of mathematical (and other) expressions are possible using current technology. Much of this can be demonstrated now.

3. Acknowledgements

This work been supported in part by DARPA, DOE, and the Systems Development Foundation.

8. Statement of Work

8.1. Operating Systems, Distributed Computing, and Programming Systems

- We will implement the interprocess communication and large file access enhancements to UNIX and make them available as part of the Berkeley Software Distributions. We will have a substantially complete experimental version of the system with the large file access enhancements documented in a technical report by March 1982. We will have a substantially complete experimental version of the system with the interprocess control enhancements documented in a technical report by June 1982. We will have a complete system including the large file access and interprocess control enhancements ready for distribution by September 1982.
- We will implement a UNIX-based distributed computing environment in the context of a network of personal workstations and larger computers not necessarily under a common administration. We will create experimental versions of various components of such an environment and document them with technical reports throughout the contract period. We will have an experimental version of the distributed system documented in a technical report by March 1983. We will have an initial version of a distributed system ready for distribution by September 1983.
- We will construct a table driven code generator which takes input from the first pass of the Portable C Compiler, the Fortran 77 compiler and the Berkeley Pascal compiler. A technical report describing the implementation and comparing its output to that of the existing compilers will be provided by September 1982. We will explore techniques for improving the generated code, implement those which appear best, and examine their impact in another technical report by September 1983.

- We will explore basic issues related to distributed computing throughout the contract period and document our research results in technical reports.

8.2. Information Management in Design and Decision Support Systems

- We will investigate how to extend the class of data representations which can be processed by a relational database system. We will extend INGRES to allow multiple representations of data items using the techniques of descriptor based access methods and will document the result in a technical report by September 1982. We will introduce the notion of bins into INGRES to provide efficient processing of spatial data; the bins will be implemented using a generalization of secondary indices and will be documented by a technical report by March 1983. We will investigate using a relational database system as an AI programming tool by experimentally rewriting some existing AI programs to use a version of INGRES which has been enhanced to allow storing information which would have been stored using Lisp; the experiment will be described in a technical report by September 1983.
- We will explore the use of forms as an efficient interface for developing various applications of database systems. The specification of a form application development system will be provided as a technical report by June 1982. A prototype system will be developed and documented in a technical report by December 1982. During the remainder of the contract period we will build a variety of applications using the prototype system in order to evaluate its interface.

8.3. Interfaces and Graphics

- We will explore connection-based style of design including how to represent and manipulate connections graphically, how to hide the details of complex connections using the concept of bundles, and how to deal with geometrical constraints. We will measure relevant aspects of existing designs and design tools to provide a context for this research. These measurements will be documented in a technical report by June 1982. We will develop a simple connection-based design

system and describe it in a technical report by June 1983.

- We will study and build a prototype mathematical software environment based on workstations which communicate with remote computers. The workstations will be graphics based and will provide the user with an integrated interface. The large computer will provide a large scale algebraic/numerical computation environment for effective problem solving. The user interface to be provided by the workstation will be designed and spelled out in a technical report by September 1982. A working user interface to Macsyma provided via a workstation will be documented in a technical report by March 1983 and a system with interactive and graphical enhancements will be documented in a technical report by September 1983.
- We will conduct both theoretical and experimental research into the applicability of Beta-splines for curve and surface representation in computer graphics systems which allow the representation and modification of geometrical shapes. A basic experimental graphics facility for use from within the UNIX environment will be constructed and documented in a technical report by September 1982. A technical report evaluating subdivision techniques for Beta-splines will be provided by September 1983.

AN IMPLEMENTATION OF HYPOTHETICAL RELATIONS

by

John Woodfill and Michael Stonebraker

Memorandum No. UCB/ERL M83/2

14 January 1983

ELECTRONICS RESEARCH LABORATORY

IMPLEMENTATION OF RULES IN RELATIONAL DATA BASE SYSTEMS

by
Michael Stonebraker, John Woodfill and Erika Andersen

Dept of Electrical Engineering and Computer Science
University of California
Berkeley, Ca.

ABSTRACT

This paper contains a proposed implementation of a rules system in a relational data base system. Such a rules system can provide data base services including integrity control, protection, alerters, triggers, and view processing. Moreoever, it can be used for user specified rules. The proposed implementation makes efficient use of an abstract data type facility by introducing new data types which assist with rule specification and enforcement.

I INTRODUCTION

Rules systems have been used extensively in Artificial Intelligence applications and are a central theme in most expert systems such as Mycin [SHOR76] and Prospector [DUDA78]. In this environment knowledge is represented as rules, typically in a first order logic representation. Hence, the data base for an expert system consists of a collection of logic formulas. The role of the data manager is to discover what rules are applicable at a given time and then to apply them. Stated differently, the data manager is largely an inference engine.

On the other hand, data base management systems have tended to represent all knowledge as pure data. The data manager is largely a collection of heuristic search procedures for finding qualifying data. Representation of first order logic statements and inference on data in the data base are rarely attempted in production data base management systems.

The purpose of this paper is to make a modest step in the direction of supporting logic statements in a data base management system. One could make this step by simply adding an inference engine to a general purpose DBMS. However, this would entail a large amount of code with no practical interaction with the current search code of a data base system. As a result, the DBMS would get much larger and would contain two essentially non overlapping subsystems. On the other hand, we strive for an implementation which integrates rules into DBMS facilities so that current search logic can be employed to control the activation of rules.

The rules system that we plan to implement is a variant of the proposal in [STON82], which was capable of expressing integrity constraints, views and protection as well as simple triggers and alarms for the relational DBMS INGRES [STON76]. Rules are of the form:

on condition
then action

The conditions which were specified include:

the type of command being executed (e.g. replace, append)

the relation affected (e.g. employee, dept)
the user issuing the command
the time of day
the day of week
the fields being updated (e.g. salary)
the fields specified in the qualification
the qualification present in the user command

The actions which we proposed included:

sending a message to a user
aborting the command
executing the command
modifying the command by adding qualification or
changing the relation names or field names

Unfortunately, these conditions and actions often affect the command which the user submitted. As such, they appear to require code that manipulates the syntax and semantics of relational commands. This string processing code appears to be complex and has little function in common with other data base facilities. In this paper we make use of two novel constructs which make implementing rules a modest undertaking. These are:

1) the notion of executing the data
and
2) a sequence of QUEL commands as a data type for a relational data base system

The remainder of this paper is organized as follows. In Section II we indicate the new data types which must be implemented and the operations required for them. Then in Section III we discuss the structural extensions to a relational data base system that will support rules execution. Lastly, Section IV and V contains some examples and our conclusions.

II RULES AS ABSTRACT DATA TYPES

Using current INGRES facilities [FOGG82, ONG82, STON82a] new data types for columns of a relation can be defined and operators on these new types specified. We use this facility to define several new types of columns and their associated operators in this section.

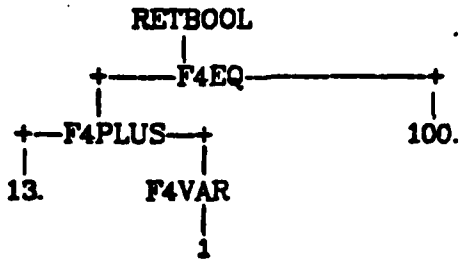
The first data type is a QUEL command, e.g.

```
range of e is employee
replace e(salary = 1.1*e.salary) where e.name = "John"
```

The abstract data type facility supports an external representation such as that above for a given data type. Moreover, when an object of the given type is stored in the data base it is converted to an internal representation. QUEL commands are converted by the INGRES parser to a parse tree representation such as the one noted in Figure 1 for the qualification "where 13. + employee.salary = 100". Consequently, a natural internal form for an object of type QUEL is a parse tree. Each node in this parse tree contains a value (e.g. 13.) and a type (e.g. floating point constant).

The second new data type which will be useful is an ATTRIBUTE-FUNCTION. This is a notion in the QUEL grammar and stands for anything that can be evaluated to a constant or the name of a column. Examples of attribute functions include:

13.



The Parse Tree for the Qualification
Where 13. + employee.salary = 100

Figure 1

1.1*employee.salary +20

newsal

The external representation is the same string format used for objects of type QUEL; the internal representation is that of a parse tree.

Two other data types of lesser significance are also needed, a TIME data type to contain a time of day value and a COMMAND data type to contain a value which is one of the QUEL commands.

Current built-in INGRES operators (e.g. *, /, +, etc.) must be extended for use with attribute functions. In addition, two new operators are also required. First, we need a function new() which will operate with integer data types. When called, it will return a new unique identifier which has not been previously used. Second, we require a partial match operator, ~, which will operate on a variety of data types and provide either equality match or match the value "*".

III INGRES CHANGES

We expect to create two rules relation, RULES1 and RULES2, with the following fields:

```
create RULES1(
    rule-id = i4,
    user-id = c10,
    time    = time,
    command = command,
    relation = c12,
    terminal = c2,
    action = quel)

create RULES2 (
    rule-id = i4,
    type = c10,
    att-fn1 = attribute-function
    operator = c5,
    att-fn2 = attribute-function)
```

For example, we might wish a rule that would add a record to an audit trail

whenever the user "Mike" updated the employee relation. This requires a row in RULES1 specified as follows:

```
append to RULES1(
    rule-id  = new(),
    user-id  = "Mike",
    command  = "replace",
    relation = "employee",
    action   = QUEL command to perform audit)
```

If additionally we wished to perform the audit action only when Mike updated the employee relation with a command containing the clause "where employee.name = "Fred"" we would add an additional tuple to RULES2 as follows:

```
append to RULES2(
    rule-id  = the one assigned in RULES1
    type     = "where"
    att-fn1  = "employee.name"
    operator = "="
    att.fn2  = "Fred")
```

We also require the possibility of executing data in the data base. We propose the following syntax:

```
range of r is relation
execute (r.field) where r.qualification
```

In this case the value of r.field must be an executable QUEL command and thereby of data type QUEL. To execute the rule that was just appended to R1 we could type:

```
range of r is R1
execute (r.action) where r.user-id = "Mike" and
                         r.command = "replace" and
                         r.relation = "employee"
```

When a QUEL command is entered by a user, it is parsed into an internal parse tree format and stored in a temporary data structure. We expect to change that data structure to be the following two main memory relations:

```
create QUERY1(
    user-id = c10,
    command = command,
    relation = c12,
    time    = time,
    terminal= c2)

create QUERY2(
    clause-id = i4,
    type      = c10,
    att-fn1   = attribute-function,
    operator  = c5,
    att-fn2   = attribute-function)
```

If the user types the query:

```
range of e is employee
retrieve (e.salary)
      where (e.name = "Mike" or e.name = "Sally")
        and e.salary > 30000
```

then INGRES will build QUERY1 to contain a single tuple with values:

QUERY1				
user-id	command	relation	time	terminal
current-user	retrieve	employee	current-time	current-terminal

QUERY2 will have four tuples as follows:

QUERY2				
clause-id	type	att-fn1	operator	att-fn2
id-x	target-list	employee.salary	=	employee.salary
id-y	where	employee.name	=	Mike
id-y	where	employee.name	=	Sally
id-z	where	employee.salary	>	30000

Notice that QUERY1 and QUERY2 contain a relational representation of the parse tree corresponding to the incoming query from the user. The where clause of the query is stored in conjunctive normal form, so that atomic formulae which are part of a disjunction have the same clause-id, while the atomic formulae and disjunctions in the conjunction have different clause-ids.

Then we execute the QUEL commands in Figure 2 to identify and execute the rules which are appropriate to the incoming command. These commands are performed by the normal INGRES search logic. Activating the rules system simply means running these commands prior to executing the user submitted command. After running the commands of Figure 2, the query is converted back to a parse tree representation and executed. Notice that the action part of a rule can update QUERY1 and QUERY2; hence modification of the user command is easily accomplished. The examples in the next section illustrate several uses for this feature:

```
range of r1 is RULES1
range of r2 is RULES2
range of q1 is QUERY1
range of q2 is QUERY2
retrieve into TEMP(r1.id, r1.quel) where
        r1.user-id ~ q1.user-id and
        r1.command ~ q1.command and
        r1.time    ~ q1.time    and
        r1.terminal~ q1.terminal

range of t is TEMP
execute (t.quel) where t.id < 0 or
       (t.id = r2.rule-id and
        set(r2.all-but-rule-id by r2.rule-id)
      = set(r2.all-but-clause-id by r2.rule-id
        where r2.all-but-rule-id ~ q2.all-but-clause-id))
```

Rule Activation in QUEL
Figure 2.

The set functions are as defined in [HELD75]. The conditions for activating a rule are:

(i) its tuple in RULES1 matches the tuple in QUERY1

and either
(ii) each tuple for the rule in RULES2 matches a tuple in QUERY2.
or
(iii) there are no required matches in RULES2
(represented by rule-id < 0).

The second condition provides appropriate rule activation when both the user query and the rule do not contain the boolean operator OR. However, a rule which should be activated when two clauses A and B are true will have two tuples in RULES2. This rule will be activated by a user query containing clauses matching A and B connected by any boolean operator. Under study is a more sophisticated activation system which will avoid this drawback.

The commands in Figure 2 cannot be executed directly because set functions have never been implemented in INGRES. Hence, we turn now to a proposed implementation of these functions.

Suppose we define a new operator "|" to be bitwise OR, and "bitor()" to be an aggregate function which bitwise ORs all qualifying fields. Then if we add the attribute "mask" to RULES2, and give each tuple for a particular rule a unique bit, the following query is correct:

```
range of t is TEMP
execute (t.quel) where t.id < 0 or
        (t.id = r2.rule-id and
     bitor(r2.mask by r2.rule-id)
     = bitor(r2.mask by r2.rule-id
        where r2.all-but-rule-id ^ q2.all-but-clause-id))
```

This solution will be quite slow, since the test for each rule involves processing a complicated aggregate. A more efficient solution involves generating masks for all rules in parallel and writing special search code as follows:

```
range of r1 is RULES1
range of r2 is RULES2
range of q1 is QUERY1
range of q2 is QUERY2
retrieve into TEMP(r1.id, r1.quel, mask = 0)  where
          r1.user-id ^ q1.user-id and
          r1.command ^ q1.command and
          r1.time    ^ q1.time    and
          r1.terminal^ q1.terminal

range of t is TEMP

foreach q2 do begin
   replace t (mask = t.mask | r2.mask)
          where t.id = r2.rule-id and
          r2.all-but-rule-id ^ q2.all-but-clause-id
end foreach

execute (t.quel) where t.id < 0 or
              (t.id = r2.rule-id and
          bitor(r2.mask by r2.rule-id)
             = t.mask)
```

Since the value of "bitor(r2.mask by r2.ruleid)" remains constant, the performance of this algorithm can be further improved by including the value of "bitor(r2.mask by r2.ruleid)" in RULES1 and copying it into TEMP as the "acceptmask". The third query would then become:

```
execute (t.quel) where t.id = r2.rule-id and
        t.acceptmask = t.mask
```

Notice the case where there are no tuples in RULES2 for a particular rule is handled with an acceptmask of zero.

Either a variable length abstract data type "bitstring" or a four byte integer can be used to store the mask. The abstract data type solution has the advantage of allowing an unlimited number of conditions for specifying rule activation, while the four byte integer solution has the advantage of simplicity and speed, but can only represent 32 conditions.

IV EXAMPLES

We give a few examples of the utility of the above constructs in this section. First, we can store a command in the data base as follows:

```
append to storedqueries (id = 6,
        quel = "range of e is employee
             retrieve (e.salary)
             where e.name = "John"")
```

We can execute the stored command by:

```
range of s is storedqueries
execute (s.quel) where s.id = 6
```

The following two examples will pertain to the query:

```
range of e is employee
replace e(salary = salary*1.5) where e.name = "Erika"
```

To represent this query INGRES will append the following tuples to the QUERY1 and QUERY2 relations:

QUERY1				
user-id	command	relation	time	terminal
current-user	replace	employee	current-time	current-terminal

QUERY2				
clause-id	type	att-fn1	operator	att-fn2
id-z	target-list	employee.salary	=	employee.salary*1.5
id-x	where	employee.name	=	Erika

Suppose we want to implement the integrity contraint to insure that employee salaries never exceed $30,000. Using query modification [STON75] we would add the clause "and employee.salary*1.5 <= 30000". to the user's qualification with the following rule:

```
append to RULES1(
    rule-id = new(), (call it id-y)
    user-id = *,     (matches any user-id)
    command = "replace",
    relation = "employee",
    action  = "range of Q2 is QUERY2
              append to QUERY2(
               clause-id = id-x,
                    type     = "where",
                    att-fn1  = Q2.att-fn2,
                    operator = "<=",
```

```
                    att-fn2  = "30000")
                where Q2.att-fn1 = "employee.salary")"
append to RULES2(
    rule-id  = id-y,
    type     = "target-list",
    att-fn1  = "employee.salary",
    operator = "=",
    att-fn2  = *)
```

Consider a transition integrity constraint that specifies that the maximum salary increase is 20%. This means that the new salary divided by the old salary must be less than or equal to 1.2. This can be achieved by appending a single tuple to R1:

```
append to RULES1(
    rule-id  = new(),
    user-id  = *,
    command  = "replace",
    relation = "employee",
    action   = "range of Q2 is QUERY2
               append to QUERY2(
                clause-id = id-x,
                    type     = "where",
                    att-fn1  = Q2.att-fn2/Q2.att-fn1,
                    operator = "<="
                    att-fn2  = "1.2")
           where Q2.att-fn1 = "employee.salary""
```

As a last example of an integrity constraint, consider a referential constraint that a new employee must be assigned to an existing department. Such a rule would be applied, for example, to the following query:

```
append to employee (name="Chris", dept = "Toy", mgr = "Ellen")
```

The corresponding tuples in QUERY2 would look like:

QUERY2				
clause-id	type	att-fn1	operator	att-fn2
id-z	target-list	employee.name	=	Chris
id-z	target-list	employee.dept	=	Toy
id-z	target-list	employee.mgr	=	Ellen

Implementation of the constraint requires checking that the department given in the target list of the append appears in the department relation. This is accomplished with the following rule:

```
append to RULES1(
    rule-id  = new(),
    user-id  = *,
    command  = "append",
    relation = "employee",
    action   = "range of Q2 is QUERY2
               append to QUERY2(
                    clause-id = id-z,
                    type     = "where",
                    att-fn1  = "dept.name",
                    operator = "=",
                    att-fn2  = Q2.att-fn2)
```

where Q2.att-fn1 = employee.dept"

Lastly, protection is achieved primarily by making use of the RULE1 relation, which pertains to the query "bookkeeping" information. Suppose we wanted to ensure that no one could access the employee relation after- hours (after 5PM and before 8AM). The following tuple would be added to the R1 relation:

```
append to RULES1(
    rule-id  = new(),
    user-id  = *,
    time     = "17:01 - 7:59",
    command  = *,
    relation = "employee",
    terminal = *,
    action   = "range of Q1 is QUERY1
               range of Q2 is QUERY2
               delete Q1
               delete Q2
```

If the query meets the conditions, the action removes the tuples in QUERY1 and QUERY2 and thereby aborts the command.

V CONCLUSIONS

This paper has presented an initial sketch of a rules system that can be embedded in a Relational DBMS. There are two potentially very powerful features to our proposal. First, it can provide a comprehensive trigger and alerter system. Real time data base applications, especially those associated with sensor data acquisition, need such a facility. Second, it provides stored DBMS commands and the possibility of parallel execution of triggered actions. In a multiprocessor environment such parallelism can be exploited.

There are also several deficiencies to the current proposal, including:

a) Rule specification is extremely complex. This could be avoided by a language processor which accepted a friendlier syntax and translated it into the one in this paper.

b) The result of the execution of a collection of rules can depend on the order in which they are activated. This is unsettling in a relational environment.

c) Rules trigger on syntax alone. For example, if we want a rule that becomes activated whenever John's employee record is affected, we trigger on any query having "employee.name = John" in the where clause. However if the incoming query is to update all employees' salaries, this rule would not be triggered.

d) Commands with multiple range variables over the same relation, so called reflexive joins, are not correctly processed by the rules engine.

e) Aggregate functions have not yet been considered.

f) As noted earlier, boolean OR is not treated correctly.

We are attempting to resolve these difficulties with further work.

REFERENCES

[DUDA78] Duda, R. et. al., "Development of the Prospector Consultation System for Mineral Exploration," SRI International, October 1978.

[FOGG82] Fogg, D., "Implementation of Domain Abstraction in the Relational Database System, INGRES", Masters Report, EECS Dept, University of California, Berkeley, CA Sept. 1982.

[HELD75] Held, G., et. al., "INGRES: A Relational Data Base System," Proc. 1975 NCC, Anaheim, Ca., May 1975.

[ONG82] Ong, J., "The Design and Implementation of Abstract Data Types in the Relational Database System, INGRES," Masters Report, EECS Dept, University of California, Berkeley, CA Sept. 1982.

[SHOR76] Shortliffe, E., "Computer Based Medical Consultations: MYCIN," Elsevier, New York, 1976.

[STON75] Stonebraker, M., "Implementation of Integrity Constraints and Views by Query Modification," Proc. 1975 ACM-SIGMOD Conference on Management of Data, San Jose, Ca., June 1975.

[STON76] Stonebraker, M. et al., "The Design and Implementation of INGRES," TODS 2, 3, September 1976.

[STON82] Stonebraker, M., et. al., "A Rules System for a Relational Data Base System," Proc. 2nd International Conference on Databases, Jerusalem, Israel, June 1982.

[STON82a] Stonebraker, M., "Extending a DBMS with Added Semantic Knowledge," Proc. NSF Workshop on Data Semantics, Intervale N.H., May 1982. (to appear in Springer-Verlag book edited by M. Brodie)

[STON83] Stonebraker, M., et. al., "Document Processing in a Relational Data Base System," ACM TOOIS, April 1983.

A 4.2bsd Interprocess Communication Primer
DRAFT of July 27, 1983

Samuel J. Leffler

Robert S. Fabry

William N. Joy

Computer Systems Research Group
Department of Electrical Engineering and Computer Science
University of California, Berkeley
Berkeley, California 94720
(415) 642-7780

ABSTRACT

This document provides an introduction to the interprocess communication facilities included in the 4.2bsd release of the VAX* UNIX** system.

It discusses the overall model for interprocess communication and introduces the interprocess communication primitives which have been added to the system. The majority of the document considers the use of these primitives in developing applications. The reader is expected to be familiar with the C programming language as all examples are written in C.

* DEC and VAX are trademarks of Digital Equipment Corporation.
** UNIX is a Trademark of Bell Laboratories.

1. INTRODUCTION

One of the most important parts of 4.2bsd is the interprocess communication facilities. These facilities are the result of more than two years of discussion and research. The facilities provided in 4.2bsd incorporate many of the ideas from current research, while trying to maintain the UNIX philosophy of simplicity and conciseness. It is hoped that the interprocess communication facilities included in 4.2bsd will establish a standard for UNIX. From the response to the design, it appears many organizations carrying out work with UNIX are adopting it.

UNIX has previously been very weak in the area of interprocess communication. Prior to the 4.2bsd facilities, the only standard mechanism which allowed two processes to communicate were pipes (the mpx files which were part of Version 7 were experimental). Unfortunately, pipes are very restrictive in that the two communicating processes must be related through a common ancestor. Further, the semantics of pipes makes them almost impossible to maintain in a distributed environment.

Earlier attempts at extending the ipc facilities of UNIX have met with mixed reaction. The majority of the problems have been related to the fact these facilities have been tied to the UNIX file system; either through naming, or implementation. Consequently, the ipc facilities provided in 4.2bsd have been designed as a totally independent subsystem. The 4.2bsd ipc allows processes to rendezvous in many ways. Processes may rendezvous through a UNIX file system-like name space (a space where all names are path names) as well as through a network name space. In fact, new name spaces may be added at a future time with only minor changes visible to users. Further, the communication facilities have been extended to included more than the simple byte stream provided by a pipe-like entity. These extensions have resulted in a completely new part of the system which users will need time to familiarize themselves with. It is likely that as more use is made of these facilities they will be refined; only time will tell.

The remainder of this document is organized in four sections. Section 2 introduces the new system calls and the basic model of communication. Section 3 describes some of the supporting library routines users may find useful in constructing distributed applications. Section 4 is concerned with the client/server model used in developing applications and includes examples of the two major types of servers. Section 5 delves into advanced topics which sophisticated users are likely to encounter when using the ipc facilities.

2. BASICS

The basic building block for communication is the *socket*. A socket is an endpoint of communication to which a name may be *bound*. Each socket in use has a *type* and one or more associated processes. Sockets exist within *communication domains*. A communication domain is an abstraction introduced to bundle common properties of processes communicating through sockets. One such property is the scheme used to name sockets. For example, in the UNIX communication domain sockets are named with UNIX path names; e.g. a socket may be named "*/dev/foo*". Sockets normally exchange data only with sockets in the same domain (it may be possible to cross domain boundaries, but only if some translation process is performed). The 4.2bsd ipc supports two separate communication domains: the UNIX domain, and the Internet domain is used by processes which communicate using the the DARPA standard communication protocols. The underlying communication facilities provided by these domains have a significant influence on the internal system implementation as well as the interface to socket facilities available to a user. An example of the latter is that a socket "operating" in the UNIX domain sees a subset of the possible error conditions which are possible when operating in the Internet domain.

2.1. Socket types

Sockets are typed according to the communication properties visible to a user. Processes are presumed to communicate only between sockets of the same type, although there is nothing that prevents communication between sockets of different types should the underlying communication protocols support this.

Three types of sockets currently are available to a user. A *stream* socket provides for the bidirectional, reliable, sequenced, and unduplicated flow of data without record boundaries. Aside from the bidirectionality of data flow, a pair of connected stream sockets provides an interface nearly identical to that of pipes*.

A *datagram* socket supports bidirectional flow of data which is not promised to be sequenced, reliable, or unduplicated. That is, a process receiving messages on a datagram socket may find messages duplicated, and, possibly, in an order different from the order in which it was sent. An important characteristic of a datagram socket is that record boundaries in data are preserved. Datagram sockets closely model the facilities found in many contemporary packet switched networks such as the Ethernet.

A *raw* socket provides users access to the underlying communication protocols which support socket abstractions. These sockets are normally datagram oriented, though their exact characteristics are dependent on the interface provided by the protocol. Raw sockets are not intended for the general user; they have been provided mainly for those interested in developing new communication protocols, or for gaining access to some of the more esoteric facilities of an existing protocol. The use of raw sockets is considered in section 5.

Two potential socket types which have interesting properties are the *sequenced packet* socket and the *reliably delivered message* socket. A sequenced packet socket is identical to a stream socket with the exception that record boundaries are preserved. This interface is very similar to that provided by the Xerox NS Sequenced Packet protocol. The reliably delivered message socket has similar properties to a datagram socket, but with reliable delivery. While these two socket types have been loosely defined, they are currently unimplemented in 4.2bsd. As such, in this document we will concern ourselves only with the three socket types for which support exists.

* In the UNIX domain, in fact, the semantics are identical and, as one might expect, pipes have been implemented internally as simply a pair of connected stream sockets.

2.2. Socket creation

To create a socket the *socket* system call is used:

```
s = socket(domain, type, protocol);
```

This call requests that the system create a socket in the specified *domain* and of the specified *type*. A particular protocol may also be requested. If the protocol is left unspecified (a value of 0), the system will select an appropriate protocol from those protocols which comprise the communication domain and which may be used to support the requested socket type. The user is returned a descriptor (a small integer number) which may be used in later system calls which operate on sockets. The domain is specified as one of the manifest constants defined in the file *<sys/socket.h>*. For the UNIX domain the constant is AF_UNIX*; for the Internet domain AF_INET. The socket types are also defined in this file and one of SOCK_STREAM, SOCK_DGRAM, or SOCK_RAW must be specified. To create a stream socket in the Internet domain the following call might be used:

```
s = socket(AF_INET, SOCK_STREAM, 0);
```

This call would result in a stream socket being created with the TCP protocol providing the underlying communication support. To create a datagram socket for on-machine use a sample call might be:

```
s = socket(AF_UNIX, SOCK_DGRAM, 0);
```

To obtain a particular protocol one selects the protocol number, as defined within the communication domain. For the Internet domain the available protocols are defined in *<netinet/in.h>* or, better yet, one may use one of the library routines discussed in section 3, such as *getprotobyname*:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
...
pp = getprotobyname("tcp");
s = socket(AF_INET, SOCK_STREAM, pp->p_proto);
```

There are several reasons a socket call may fail. Aside from the rare occurrence of lack of memory (ENOBUFS), a socket request may fail due to a request for an unknown protocol (EPROTONOSUPPORT), or a request for a type of socket for which there is no supporting protocol (EPROTOTYPE).

2.3. Binding names

A socket is created without a name. Until a name is bound to a socket, processes have no way to reference it and, consequently, no messages may be received on it. The *bind* call is used to assign a name to a socket:

```
bind(s, name, namelen);
```

The bound name is a variable length byte string which is interpreted by the supporting protocol(s). Its interpretation may vary from communication domain to communication domain (this is one of the properties which comprise the "domain"). In the UNIX domain names are path names while in the Internet domain names contain an Internet address and port number. If one wanted to bind the name "/dev/foo" to a UNIX domain socket, the following would be used:

* The manifest constants are named AF_whatever as they indicate the "address format" to use in interpreting names.

```
bind(s, "/dev/foo", sizeof ("/dev/foo") - 1);
```

(Note how the null byte in the name is not counted as part of the name.) In binding an Internet address things become more complicated. The actual call is simple,

```
#include <sys/types.h>
#include <netinet/in.h>
...
struct sockaddr_in sin;
...
bind(s, &sin, sizeof (sin));
```

but the selection of what to place in the address *sin* requires some discussion. We will come back to the problem of formulating Internet addresses in section 3 when the library routines used in name resolution are discussed.

2.4. Connection establishment

With a bound socket it is possible to rendezvous with an unrelated process. This operation is usually asymmetric with one process a "client" and the other a "server". The client requests services from the server by initiating a "connection" to the server's socket. The server, when willing to offer its advertised services, passively "listens" on its socket. On the client side the *connect* call is used to initiate a connection. Using the UNIX domain, this might appear as,

```
connect(s, "server-name", sizeof ("server-name"));
```

while in the Internet domain,

```
struct sockaddr_in server;
connect(s, &server, sizeof (server));
```

If the client process's socket is unbound at the time of the connect call, the system will automatically select and bind a name to the socket; c.f. section 5.4. An error is returned when the connection was unsuccessful (any name automatically bound by the system, however, remains). Otherwise, the socket is associated with the server and data transfer may begin.

Many errors can be returned when a connection attempt fails. The most common are:

ETIMEDOUT
: After failing to establish a connection for a period of time, the system decided there was no point in retrying the connection attempt any more. This usually occurs because the destination host is down, or because problems in the network resulted in transmissions being lost.

ECONNREFUSED
: The host refused service for some reason. When connecting to a host running 4.2bsd this is usually due to a server process not being present at the requested name.

ENETDOWN or EHOSTDOWN
: These operational errors are returned based on status information delivered to the client host by the underlying communication services.

ENETUNREACH or EHOSTUNREACH
: These operational errors can occur either because the network or host is unknown (no route to the network or host is present), or because of status information returned by intermediate gateways or switching nodes. Many times the status returned is not sufficient to distinguish a network being down from a host being down. In these cases the system is conservative and indicates the entire network is unreachable.

For the server to receive a client's connection it must perform two steps after binding its socket. The first is to indicate a willingness to listen for incoming connection requests:

```
listen(s, 5);
```

The second parameter to the *listen* call specifies the maximum number of outstanding connections which may be queued awaiting acceptance by the server process. Should a connection be requested while the queue is full, the connection will not be refused, but rather the individual messages which comprise the request will be ignored. This gives a harried server time to make room in its pending connection queue while the client retries the connection request. Had the connection been returned with the ECONNREFUSED error, the client would be unable to tell if the server was up or not. As it is now it is still possible to get the ETIMEDOUT error back, though this is unlikely. The backlog figure supplied with the listen call is limited by the system to a maximum of 5 pending connections on any one queue. This avoids the problem of processes hogging system resources by setting an infinite backlog, then ignoring all connection requests.

With a socket marked as listening, a server may *accept* a connection:

```
fromlen = sizeof (from);
snew = accept(s, &from, &fromlen);
```

A new descriptor is returned on receipt of a connection (along with a new socket). If the server wishes to find out who its client is, it may supply a buffer for the client socket's name. The value-result parameter *fromlen* is initialized by the server to indicate how much space is associated with *from*, then modified on return to reflect the true size of the name. If the client's name is not of interest, the second parameter may be zero.

Accept normally blocks. That is, the call to accept will not return until a connection is available or the system call is interrupted by a signal to the process. Further, there is no way for a process to indicate it will accept connections from only a specific individual, or individuals. It is up to the user process to consider who the connection is from and close down the connection if it does not wish to speak to the process. If the server process wants to accept connections on more than one socket, or not block on the accept call there are alternatives; they will be considered in section 5.

2.5. Data transfer

With a connection established, data may begin to flow. To send and receive data there are a number of possible calls. With the peer entity at each end of a connection anchored, a user can send or receive a message without specifying the peer. As one might expect, in this case, then the normal *read* and *write* system calls are useable,

```
write(s, buf, sizeof (buf));
read(s, buf, sizeof (buf));
```

In addition to *read* and *write*, the new calls *send* and *recv* may be used:

```
send(s, buf, sizeof (buf), flags);
recv(s, buf, sizeof (buf), flags);
```

While *send* and *recv* are virtually identical to *read* and *write*, the extra *flags* argument is important. The flags may be specified as a non-zero value if one or more of the following is required:

SOF_OOB	send/receive out of band data
SOF_PREVIEW	look at data without reading
SOF_DONTROUTE	send data without routing packets

Out of band data is a notion specific to stream sockets, and one which we will not immediately consider. The option to have data sent without routing applied to the outgoing packets is currently used only by the routing table management process, and is unlikely to be of interest to the casual user. The ability to preview data is, however, of interest. When SOF_PREVIEW is specified with a *recv* call, any data present is returned to the user, but treated as still

"unread". That is, the next *read* or *recv* call applied to the socket will return the data previously previewed.

2.6. Discarding sockets

Once a socket is no longer of interest, it may be discarded by applying a *close* to the descriptor,

```
close(s);
```

If data is associated with a socket which promises reliable delivery (e.g. a stream socket) when a close takes place, the system will continue to attempt to transfer the data. However, after a fairly long period of time, if the data is still undelivered, it will be discarded. Should a user have no use for any pending data, it may perform a *shutdown* on the socket prior to closing it. This call is of the form:

```
shutdown(s, how);
```

where *how* is 0 if the user is no longer interested in reading data, 1 if no more data will be sent, or 2 if no data is to be sent or received. Applying shutdown to a socket causes any data queued to be immediately discarded.

2.7. Connectionless sockets

To this point we have been concerned mostly with sockets which follow a connection oriented model. However, there is also support for connectionless interactions typical of the datagram facilities found in contemporary packet switched networks. A datagram socket provides a symmetric interface to data exchange. While processes are still likely to be client and server, there is no requirement for connection establishment. Instead, each message includes the destination address.

Datagram sockets are created as before, and each should have a name bound to it in order that the recipient of a message may identify the sender. To send data, the *sendto* primitive is used,

```
sendto(s, buf, buflen, flags, &to, tolen);
```

The *s*, *buf*, *buflen*, and *flags* parameters are used as before. The *to* and *tolen* values are used to indicate the intended recipient of the message. When using an unreliable datagram interface, it is unlikely any errors will be reported to the sender. Where information is present locally to recognize a message which may never be delivered (for instance when a network is unreachable), the call will return −1 and the global value *errno* will contain an error number.

To receive messages on an unconnected datagram socket, the *recvfrom* primitive is provided:

```
recvfrom(s, buf, buflen, flags, &from, &fromlen);
```

Once again, the *fromlen* parameter is handled in a value-result fashion, initially containing the size of the *from* buffer.

In addition to the two calls mentioned above, datagram sockets may also use the *connect* call to associate a socket with a specific address. In this case, any data sent on the socket will automatically be addressed to the connected peer, and only data received from that peer will be delivered to the user. Only one connected address is permitted for each socket (i.e. no multicasting). Connect requests on datagram sockets return immediately, as this simply results in the system recording the peer's address (as compared to a stream socket where a connect request initiates establishment of an end to end connection). Other of the less important details of datagram sockets are described in section 5.

2.8. Input/Output multiplexing

One last facility often used in developing applications is the ability to multiplex i/o requests among multiple sockets and/or files. This is done using the *select* call:

```
select(nfds, &readfds, &writefds, &execptfds, &timeout);
```

Select takes as arguments three bit masks, one for the set of file descriptors for which the caller wishes to be able to read data on, one for those descriptors to which data is to be written, and one for which exceptional conditions are pending. Bit masks are created by or-ing bits of the form "1 << *fd*". That is, a descriptor *fd* is selected if a 1 is present in the *fd*'th bit of the mask. The parameter *nfds* specifies the range of file descriptors (i.e. one plus the value of the largest descriptor) specified in a mask.

A timeout value may be specified if the selection is not to last more than a predetermined period of time. If *timeout* is set to 0, the selection takes the form of a *poll*, returning immediately. If the last parameter is a null pointer, the selection will block indefinitely*. *Select* normally returns the number of file descriptors selected. If the *select* call returns due to the timeout expiring, then a value of −1 is returned along with the error number EINTR.

Select provides a synchronous multiplexing scheme. Asynchronous notification of output completion, input availability, and exceptional conditions is possible through use of the SIGIO and SIGURG signals described in section 5.

* To be more specific, a return takes place only when a descriptor is selectable, or when a signal is received by the caller, interrupting the system call.

3. NETWORK LIBRARY ROUTINES

The discussion in section 2 indicated the possible need to locate and construct network addresses when using the interprocess communication facilities in a distributed environment. To aid in this task a number of routines have been added to the standard C run-time library. In this section we will consider the new routines provided to manipulate network addresses. While the 4.2bsd networking facilities support only the DARPA standard Internet protocols, these routines have been designed with flexibility in mind. As more communication protocols become available, we hope the same user interface will be maintained in accessing network-related address data bases. The only difference should be the values returned to the user. Since these values are normally supplied the system, users should not need to be directly aware of the communication protocol and/or naming conventions in use.

Locating a service on a remote host requires many levels of mapping before client and server may communicate. A service is assigned a name which is intended for human consumption; e.g. "the *login server* on host monet". This name, and the name of the peer host, must then be translated into network *addresses* which are not necessarily suitable for human consumption. Finally, the address must then used in locating a physical *location* and *route* to the service. The specifics of these three mappings is likely to vary between network architectures. For instance, it is desirable for a network to not require hosts be named in such a way that their physical location is known by the client host. Instead, underlying services in the network may discover the actual location of the host at the time a client host wishes to communicate. This ability to have hosts named in a location independent manner may induce overhead in connection establishment, as a discovery process must take place, but allows a host to be physically mobile without requiring it to notify its clientele of its current location.

Standard routines are provided for: mapping host names to network addresses, network names to network numbers, protocol names to protocol numbers, and service names to port numbers and the appropriate protocol to use in communicating with the server process. The file *<netdb.h>* must be included when using any of these routines.

3.1. Host names

A host name to address mapping is represented by the *hostent* structure:

```
struct  hostent {
        char    *h_name;        /* official name of host */
        char    **h_aliases;    /* alias list */
        int     h_addrtype;     /* host address type */
        int     h_length;       /* length of address */
        char    *h_addr;        /* address */
};
```

The official name of the host and its public aliases are returned, along with a variable length address and address type. The routine *gethostbyname*(3N) takes a host name and returns a *hostent* structure, while the routine *gethostbyaddr*(3N) maps host addresses into a *hostent* structure. It is possible for a host to have many addresses, all having the same name. *Gethostbyname* returns the first matching entry in the data base file */etc/hosts*; if this is unsuitable, the lower level routine *gethostent*(3N) may be used. For example, to obtain a *hostent* structure for a host on a particular network the following routine might be used (for simplicity, only Internet addresses are considered):

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
...
struct hostent *
gethostbynameandnet(name, net)
        char *name;
        int net;
{
        register struct hostent *hp;
        register char **cp;

        sethostent(0);
        while ((hp = gethostent()) != NULL) {
                if (hp->h_addrtype != AF_INET)
                        continue;
                if (strcmp(name, hp->h_name)) {
                        for (cp = hp->h_aliases; cp && *cp != NULL; cp++)
                                if (strcmp(name, *cp) == 0)
                                goto found;
                        continue;
                }
        found:
                if (in_netof(*(struct in_addr *)hp->h_addr)) == net)
                        break;
        }
        endhostent(0);
        return (hp);
}
```

(*in_netof*(3N) is a standard routine which returns the network portion of an Internet address.)

3.2. Network names

As for host names, routines for mapping network names to numbers, and back, are provided. These routines return a *netent* structure:

```
/*
 * Assumption here is that a network number
 * fits in 32 bits -- probably a poor one.
 */
struct  netent {
        char    *n_name;        /* official name of net */
        char    **n_aliases;    /* alias list */
        int     n_addrtype;     /* net address type */
        int     n_net;          /* network # */
};
```

The routines *getnetbyname*(3N), *getnetbynumber*(3N), and *getnetent*(3N) are the network counterparts to the host routines described above.

3.3. Protocol names

For protocols the *protoent* structure defines the protocol-name mapping used with the routines *getprotobyname*(3N), *getprotobynumber*(3N), and *getprotoent*(3N):

```
struct  protoent {
        char    *p_name;        /* official protocol name */
        char    **p_aliases;    /* alias list */
        int     p_proto;        /* protocol # */
};
```

3.4. Service names

Information regarding services is a bit more complicated. A service is expected to reside at a specific "port" and employ a particular communication protocol. This view is consistent with the Internet domain, but inconsistent with other network architectures. Further, a service may reside on multiple ports or support multiple protocols. If either of these occurs, the higher level library routines will have to be bypassed in favor of homegrown routines similar in spirit to the "gethostbynameandnet" routine described above. A service mapping is described by the *servent* structure,

```
struct  servent {
        char    *s_name;        /* official service name */
        char    **s_aliases;    /* alias list */
        int     s_port;         /* port # */
        char    *s_proto;       /* protocol to use */
};
```

The routine *getservbyname*(3N) maps service names to a servent structure by specifying a service name and, optionally, a qualifying protocol. Thus the call

```
sp = getservbyname("telnet", (char *)0);
```

returns the service specification for a telnet server using any protocol, while the call

```
sp = getservbyname("telnet", "tcp");
```

returns only that telnet server which uses the TCP protocol. The routines *getservbyport*(3N) and *getservent*(3N) are also provided. The *getservbyport* routine has an interface similar to that provided by *getservbyname*; an optional protocol name may be specified to qualify lookups.

3.5. Miscellaneous

With the support routines described above, an application program should rarely have to deal directly with addresses. This allows services to be developed as much as possible in a network independent fashion. It is clear, however, that purging all network dependencies is very difficult. So long as the user is required to supply network addresses when naming services and sockets there will always some network dependency in a program. For example, the normal code included in client programs, such as the remote login program, is of the form shown in Figure 1. (This example will be considered in more detail in section 4.)

If we wanted to make the remote login program independent of the Internet protocols and addressing scheme we would be forced to add a layer of routines which masked the network dependent aspects from the mainstream login code. For the current facilities available in the system this does not appear to be worthwhile. Perhaps when the system is adapted to different network architectures the utilities will be reorganized more cleanly.

Aside from the address-related data base routines, there are several other routines available in the run-time library which are of interest to users. These are intended mostly to simplify manipulation of names and addresses. Table 1 summarizes the routines for manipulating variable length byte strings and handling byte swapping of network addresses and values.

The byte swapping routines are provided because the operating system expects addresses to be supplied in network order. On a VAX, or machine with similar architecture, this is usually reversed. Consequently, programs are sometimes required to byte swap quantities. The

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <stdio.h>
#include <netdb.h>
...
main(argc, argv)
        char *argv[];
{
        struct sockaddr_in sin;
        struct servent *sp;
        struct hostent *hp;
        int s;
        ...
        sp = getservbyname("login", "tcp");
        if (sp == NULL) {
                fprintf(stderr, "rlogin: tcp/login: unknown service\n");
                exit(1);
        }
        hp = gethostbyname(argv[1]);
        if (hp == NULL) {
                fprintf(stderr, "rlogin: %s: unknown host\n", argv[1]);
                exit(2);
        }
        bzero((char *)&sin, sizeof (sin));
        bcopy(hp->h_addr, (char *)&sin.sin_addr, hp->h_length);
        sin.sin_family = hp->h_addrtype;
        sin.sin_port = sp->s_port;
        s = socket(AF_INET, SOCK_STREAM, 0);
        if (s < 0) {
                perror("rlogin: socket");
                exit(3);
        }
        ...
        if (connect(s, (char *)&sin, sizeof (sin)) < 0) {
                perror("rlogin: connect");
                exit(5);
        }
        ...
}
```

Figure 1. Remote login client code.

Call	Synopsis
bcmp(s1, s2, n)	compare byte-strings; 0 if same, not 0 otherwise
bcopy(s1, s2, n)	copy n bytes from s1 to s2
bzero(base, n)	zero-fill n bytes starting at base
htonl(val)	convert 32-bit quantity from host to network byte order
htons(val)	convert 16-bit quantity from host to network byte order
ntohl(val)	convert 32-bit quantity from network to host byte order
ntohs(val)	convert 16-bit quantity from network to host byte order

Table 1. C run-time routines.

library routines which return network addresses provide them in network order so that they may simply be copied into the structures provided to the system. This implies users should encounter the byte swapping problem only when *interpreting* network addresses. For example, if an Internet port is to be printed out the following code would be required:

```
printf("port number %d\n", ntohs(sp->s_port));
```

On machines other than the VAX these routines are defined as null macros.

4. CLIENT/SERVER MODEL

The most commonly used paradigm in constructing distributed applications is the client/server model. In this scheme client applications request services from a server process. This implies an asymmetry in establishing communication between the client and server which has been examined in section 2. In this section we will look more closely at the interactions between client and server, and consider some of the problems in developing client and server applications.

Client and server require a well known set of conventions before service may be rendered (and accepted). This set of conventions comprises a protocol which must be implemented at both ends of a connection. Depending on the situation, the protocol may be symmetric or asymmetric. In a symmetric protocol, either side may play the master or slave roles. In an asymmetric protocol, one side is immutably recognized as the master, with the other the slave. An example of a symmetric protocol is the TELNET protocol used in the Internet for remote terminal emulation. An example of an asymmetric protocol is the Internet file transfer protocol, FTP. No matter whether the specific protocol used in obtaining a service is symmetric or asymmetric, when accessing a service there is a "client process" and a "server process". We will first consider the properties of server processes, then client processes.

A server process normally listens at a well know address for service requests. Alternative schemes which use a service server may be used to eliminate a flock of server processes clogging the system while remaining dormant most of the time. The Xerox Courier protocol uses the latter scheme. When using Courier, a Courier client process contacts a Courier server at the remote host and identifies the service it requires. The Courier server process then creates the appropriate server process based on a data base and "splices" the client and server together, voiding its part in the transaction. This scheme is attractive in that the Courier server process may provide a single contact point for all services, as well as carrying out the initial steps in authentication. However, while this is an attractive possibility for standardizing access to services, it does introduce a certain amount of overhead due to the intermediate process involved. Implementations which provide this type of service within the system can minimize the cost of client server rendezvous. The *portal* notion described in the "4.2BSD System Manual" embodies many of the ideas found in Courier, with the rendezvous mechanism implemented internal to the system.

4.1. Servers

In 4.2bsd most servers are accessed at well known Internet addresses or UNIX domain names. When a server is started at boot time it advertises it services by listening at a well know location. For example, the remote login server's main loop is of the form shown in Figure 2.

The first step taken by the server is look up its service definition:

```
sp = getservbyname("login", "tcp");
if (sp == NULL) {
        fprintf(stderr, "rlogind: tcp/login: unknown service\n");
        exit(1);
}
```

This definition is used in later portions of the code to define the Internet port at which it listens for service requests (indicated by a connection).

Step two is to disassociate the server from the controlling terminal of its invoker. This is important as the server will likely not want to receive signals delivered to the process group of the controlling terminal.

```
main(argc, argv)
        int argc;
        char **argv;
{
        int f;
        struct sockaddr_in from;
        struct servent *sp;

        sp = getservbyname("login", "tcp");
        if (sp == NULL) {
                fprintf(stderr, "rlogind: tcp/login: unknown service\n");
                exit(1);
        }
        ...
#ifndef DEBUG
        <<disassociate server from controlling terminal>>
#endif
        ...
        sin.sin_port = sp->s_port;
        ...
        f = socket(AF_INET, SOCK_STREAM, 0);
        ...
        if (bind(f, (caddr_t)&sin, sizeof (sin)) < 0) {
                ...
        }
        ...
        listen(f, 5);
        for (;;) {
                int g, len = sizeof (from);

                g = accept(f, &from, &len);
                if (g < 0) {
                        if (errno != EINTR)
                                perror("rlogind: accept");
                        continue;
                }
                if (fork() == 0) {
                        close(f);
                        doit(g, &from);
                }
                close(g);
        }
}
```

Figure 2. Remote login server.

Once a server has established a pristine environment, it creates a socket and begins accepting service requests. The *bind* call is required to insure the server listens at its expected location. The main body of the loop is fairly simple:

```
for (;;) {
        int g, len = sizeof (from);

        g = accept(f, &from, &len);
        if (g < 0) {
                if (errno != EINTR)
                        perror("rlogind: accept");
                continue;
        }
        if (fork() == 0) {
                close(f);
                doit(g, &from);
        }
        close(g);
}
```

An *accept* call blocks the server until a client requests service. This call could return a failure status if the call is interrupted by a signal such as SIGCHLD (to be discussed in section 5). Therefore, the return value from *accept* is checked to insure a connection has actually been established. With a connection in hand, the server then forks a child process and invokes the main body of the remote login protocol processing. Note how the socket used by the parent for queueing connection requests is closed in the child, while the socket created as a result of the accept is closed in the parent. The address of the client is also handed the *doit* routine because it requires it in authenticating clients.

4.2. Clients

The client side of the remote login service was shown earlier in Figure 1. One can see the separate, asymmetric roles of the client and server clearly in the code. The server is a passive entity, listening for client connections, while the client process is an active entity, initiating a connection when invoked.

Let us consider more closely the steps taken by the client remote login process. As in the server process the first step is to locate the service definition for a remote login:

```
sp = getservbyname("login", "tcp");
if (sp == NULL) {
        fprintf(stderr, "rlogin: tcp/login: unknown service\n");
        exit(1);
}
```

Next the destination host is looked up with a *gethostbyname* call:

```
hp = gethostbyname(argv[1]);
if (hp == NULL) {
        fprintf(stderr, "rlogin: %s: unknown host\n", argv[1]);
        exit(2);
}
```

With this accomplished, all that is required is to establish a connection to the server at the requested host and start up the remote login protocol. The address buffer is cleared, then filled in with the Internet address of the foreign host and the port number at which the login process resides:

```
bzero((char *)&sin, sizeof (sin));
bcopy(hp->h_addr, (char *)sin.sin_addr, hp->h_length);
sin.sin_family = hp->h_addrtype;
sin.sin_port = sp->s_port;
```

A socket is created, and a connection initiated.

```
s = socket(hp->h_addrtype, SOCK_STREAM, 0);
if (s < 0) {
        perror("riogin: socket");
        exit(3);
}
...
if (connect(s, (char *)&sin, sizeof (sin)) < 0) {
        perror("riogin: connect");
        exit(4);
}
```

The details of the remote login protocol will not be considered here.

4.3. Connectionless servers

While connection-based services are the norm, some services are based on the use of datagram sockets. One, in particular, is the "rwho" service which provides users with status information for hosts connected to a local area network. This service, while predicated on the ability to *broadcast* information to all hosts connected to a particular network, is of interest as an example usage of datagram sockets.

A user on any machine running the rwho server may find out the current status of a machine with the *ruptime*(1) program. The output generated is illustrated in Figure 3.

```
arpa        up   9:45,     5 users, load  1.15,  1.39,  1.31
cad         up   2+12:04,  8 users, load  4.67,  5.13,  4.59
calder      up   10:10,    0 users, load  0.27,  0.15,  0.14
dali        up   2+06:28,  9 users, load  1.04,  1.20,  1.65
degas       up   25+09:48, 0 users, load  1.49,  1.43,  1.41
ear         up   5+00:05,  0 users, load  1.51,  1.54,  1.56
ernie     down   0:24
esvax     down   17:04
ingres    down   0:26
kim         up   3+09:16,  8 users, load  2.03,  2.46,  3.11
matisse     up   3+06:18,  0 users, load  0.03,  0.03,  0.05
medea       up   3+09:39,  2 users, load  0.35,  0.37,  0.50
merlin    down   19+15:37
miro        up   1+07:20,  7 users, load  4.59,  3.28,  2.12
monet       up   1+00:43,  2 users, load  0.22,  0.09,  0.07
oz        down   16:09
statvax     up   2+15:57,  3 users, load  1.52,  1.81,  1.86
ucbvax      up   9:34,     2 users, load  6.08,  5.16,  3.28
```

Figure 3. ruptime output.

Status information for each host is periodically broadcast by rwho server processes on each machine. The same server process also receives the status information and uses it to update a database. This database is then interpreted to generate the status information for each host. Servers operate autonomously, coupled only by the local network and its broadcast capabilities.

The rwho server, in a simplified form, is pictured in Figure 4. There are two separate tasks performed by the server. The first task is to act as a receiver of status information broadcast by other hosts on the network. This job is carried out in the main loop of the program. Packets received at the rwho port are interrogated to insure they've been sent by another rwho server process, then are time stamped with their arrival time and used to update a file indicating the status of the host. When a host has not been heard from for an extended period of time, the database interpretation routines assume the host is down and indicate such on the status reports. This algorithm is prone to error as a server may be down while a host is actually up, but serves our current needs.

```
main()
{
        ...
        sp = getservbyname("who", "udp");
        net = getnetbyname("localnet");
        sin.sin_addr = inet_makeaddr(INADDR_ANY, net);
        sin.sin_port = sp->s_port;
        ...
        s = socket(AF_INET, SOCK_DGRAM, 0);
        ...
        bind(s, &sin, sizeof (sin));
        ...
        sigset(SIGALRM, onalrm);
        onalrm();
        for (;;) {
                struct whod wd;
                int cc, whod, len = sizeof (from);

                cc = recvfrom(s, (char *)&wd, sizeof (struct whod), 0, &from, &len);
                if (cc <= 0) {
                        if (cc < 0 && errno != EINTR)
                                perror("rwhod: recv");
                        continue;
                }
                if (from.sin_port != sp->s_port) {
                        fprintf(stderr, "rwhod: %d: bad from port\n",
                                ntohs(from.sin_port));
                        continue;
                }
                ...
                if (!verify(wd.wd_hostname)) {
                        fprintf(stderr, "rwhod: malformed host name from %x\n",
                                ntohl(from.sin_addr.s_addr));
                        continue;
                }
                (void) sprintf(path, "%s/whod.%s", RWHODIR, wd.wd_hostname);
                whod = open(path, FWRONLY|FCREATE|FTRUNCATE, 0666);
                ...
                (void) time(&wd.wd_recvtime);
                (void) write(whod, (char *)&wd, cc);
                (void) close(whod);
        }
}
```

Figure 4. rwho server.

The second task performed by the server is to supply information regarding the status of its host. This involves periodically acquiring system status information, packaging it up in a message and broadcasting it on the local network for other rwho servers to hear. The supply function is triggered by a timer and runs off a signal. Locating the system status information is somewhat involved, but uninteresting. Deciding where to transmit the resultant packet does, however, indicates some problems with the current protocol.

Status information is broadcast on the local network. For networks which do not support the notion of broadcast another scheme must be used to simulate or replace broadcasting. One possibility is to enumerate the known neighbors (based on the status received). This, unfortunately, requires some bootstrapping information, as a server started up on a quiet network will have no known neighbors and thus never receive, or send, any status information. This is the identical problem faced by the routing table management process in propagating routing status information. The standard solution, unsatisfactory as it may be, is to inform one or more servers of known neighbors and request that they always communicate with these neighbors. If each server has at least one neighbor supplied it, status information may then propagate through a neighbor to hosts which are not (possibly) directly neighbors. If the server is able to support networks which provide a broadcast capability, as well as those which do not, then networks with an arbitrary topology may share status information*.

The second problem with the current scheme is that the rwho process services only a single local network, and this network is found by reading a file. It is important that software operating in a distributed environment not have any site-dependent information compiled into it. This would require a separate copy of the server at each host and make maintenance a severe headache. 4.2bsd attempts to isolate host-specific information from applications by providing system calls which return the necessary information†. Unfortunately, no straightforward mechanism currently exists for finding the collection of networks to which a host is directly connected. Thus the rwho server performs a lookup in a file to find its local network. A better, though still unsatisfactory, scheme used by the routing process is to interrogate the system data structures to locate those directly connected networks. A mechanism to acquire this information from the system would be a useful addition.

* One must, however, be concerned about "loops". That is, if a host is connected to multiple networks, it will receive status information from itself. This can lead to an endless, wasteful, exchange of information.
† An example of such a system call is the *gethostname*(2) call which returns the host's "official" name.

5. ADVANCED TOPICS

A number of facilities have yet to be discussed. For most users of the ipc the mechanisms already described will suffice in constructing distributed applications. However, others will find need to utilize some of the features which we consider in this section.

5.1. Out of band data

The stream socket abstraction includes the notion of "out of band" data. Out of band data is a logically independent transmission channel associated with each pair of connected stream sockets. Out of band data is delivered to the user independently of normal data along with the SIGURG signal. In addition to the information passed, a logical mark is placed in the data stream to indicate the point at which the out of band data was sent. The remote login and remote shell applications use this facility to propagate signals from between client and server processes. When a signal is expected to flush any pending output from the remote process(es), all data up to the mark in the data stream is discarded.

The stream abstraction defines that the out of band data facilities must support the reliable delivery of at least one out of band message at a time. This message may contain at least one byte of data, and at least one message may be pending delivery to the user at any one time. For communications protocols which support only in-band signaling (i.e. the urgent data is delivered in sequence with the normal data) the system extracts the data from the normal data stream and stores it separately. This allows users to choose between receiving the urgent data in order and receiving it out of sequence without having to buffer all the intervening data.

To send an out of band message the SOF_OOB flag is supplied to a *send* or *sendto* calls, while to receive out of band data SOF_OOB should be indicated when performing a *recvfrom* or *recv* call. To find out if the read pointer is currently pointing at the mark in the data stream, the SIOCATMARK ioctl is provided:

```
ioctl(s, SIOCATMARK, &yes);
```

If *yes* is a 1 on return, the next read will return data after the mark. Otherwise (assuming out of band data has arrived), the next read will provide data sent by the client prior to transmission of the out of band signal. The routine used in the remote login process to flush output on receipt of an interrupt or quit signal is shown in Figure 5.

5.2. Signals and process groups

Due to the existence of the SIGURG and SIGIO signals each socket has an associated process group (just as is done for terminals). This process group is initialized to the process group of its creator, but may be redefined at a later time with the SIOCSPGRP ioctl:

```
oob()
{
        int out = 1+1;
        char waste[BUFSIZ], mark;

        signal(SIGURG, oob);
        /* flush local terminal input and output */
        ioctl(1, TIOCFLUSH, (char *)&out);
        for (;;) {
                if (ioctl(rem, SIOCATMARK, &mark) < 0) {
                        perror("ioctl");
                        break;
                }
                if (mark)
                        break;
                (void) read(rem, waste, sizeof (waste));
        }
        recv(rem, &mark, 1, SOF_OOB);
        ...
}
```

Figure 5. Flushing terminal i/o on receipt of out of band data.

```
ioctl(s, SIOCSPGRP, &pgrp);
```

A similar ioctl, SIOCGPGRP, is available for determining the current process group of a socket.

5.3. Pseudo terminals

Many programs will not function properly without a terminal for standard input and output. Since a socket is not a terminal, it is often necessary to have a process communicating over the network do so through a *pseudo terminal*. A pseudo terminal is actually a pair of devices, master and slave, which allow a process to serve as an active agent in communication between processes and users. Data written on the slave side of a pseudo terminal is supplied as input to a process reading from the master side. Data written on the master side is given the slave as input. In this way, the process manipulating the master side of the pseudo terminal has control over the information read and written on the slave side. The remote login server uses pseudo terminals for remote login sessions. A user logging in to a machine across the network is provided a shell with a slave pseudo terminal as standard input, output, and error. The server process then handles the communication between the programs invoked by the remote shell and the user's local client process. When a user sends an interrupt or quit signal to a process executing on a remote machine, the client login program traps the signal, sends an out of band message to the server process who then uses the signal number, sent as the data value in the out of band message, to perform a *killpg*(2) on the appropriate process group.

5.4. Internet address binding

Binding addresses to sockets in the Internet domain can be fairly complex. Communicating processes are bound by an *association*. An association is composed of local and foreign addresses, and local and foreign ports. Port numbers are allocated out of separate spaces, one for each Internet protocol. Associations are always unique. That is, there may never be duplicate <protocol, local address, local port, foreign address, foreign port> tuples.

The bind system call allows a process to specify half of an association, <local address, local port>, while the connect and accept primitives are used to complete a socket's association. Since the association is created in two steps the association uniqueness requirement indicated above could be violated unless care is taken. Further, it is unrealistic to expect user

programs to always know proper values to use for the local address and local port since a host may reside on multiple networks and the set of allocated port numbers is not directly accessible to a user.

To simplify local address binding the notion of a "wildcard" address has been provided. When an address is specified as INADDR_ANY (a manifest constant defined in <netinet/in.h>), the system interprets the address as "any valid address". For example, to bind a specific port number to a socket, but leave the local address unspecified, the following code might be used:

```
#include <sys/types.h>
#include <netinet/in.h>
...
struct sockaddr_in sin;
...
s = socket(AF_INET, SOCK_STREAM, 0);
sin.sin_family = AF_INET;
sin.sin_addr.s_addr = INADDR_ANY;
sin.sin_port = MYPORT;
bind(s, (char *)&sin, sizeof (sin));
```

Sockets with wildcarded local addresses may receive messages directed to the specified port number, and addressed to any of the possible addresses assigned a host. For example, if a host is on a networks 46 and 10 and a socket is bound as above, then an accept call is performed, the process will be able to accept connection requests which arrive either from network 46 or network 10.

In a similar fashion, a local port may be left unspecified (specified as zero), in which case the system will select an appropriate port number for it. For example:

```
sin.sin_addr.s_addr = MYADDRESS;
sin.sin_port = 0;
bind(s, (char *)&sin, sizeof (sin));
```

The system selects the port number based on two criteria. The first is that ports numbered 0 through 1023 are reserved for privileged users (i.e. the super user). The second is that the port number is not currently bound to some other socket. In order to find a free port number in the privileged range the following code is used by the remote shell server:

```
struct sockaddr_in sin;
...
lport = IPPORT_RESERVED - 1;
sin.sin_addr.s_addr = INADDR_ANY;
...
for (;;) {
        sin.sin_port = htons((u_short)lport);
        if (bind(s, (caddr_t)&sin, sizeof (sin)) >= 0)
                break;
        if (errno != EADDRINUSE && errno != EADDRNOTAVAIL) {
                perror("socket");
                break;
        }
        lport--;
        if (lport == IPPORT_RESERVED/2) {
                fprintf(stderr, "socket: All ports in use\n");
                break;
        }
}
```

The restriction on allocating ports was done to allow processes executing in a "secure" environment to perform authentication based on the originating address and port number.

In certain cases the algorithm used by the system in selecting port numbers is unsuitable for an application. *This is due to associations being created in a two step process.* For example, the Internet file transfer protocol, FTP, specifies that data connections must always originate from the same local port. However, duplicate associations are avoided by connecting to different foreign ports. In this situation the system would disallow binding the same local address and port number to a socket if a previous data connection's socket were around. To override the default port selection algorithm then an option call must be performed prior to address binding:

```
setsockopt(s, SOL_SOCKET, SO_REUSEADDR, (char *)0, 0);
bind(s, (char *)&sin, sizeof (sin));
```

With the above call, local addresses may be bound which are already in use. This does not violate the uniqueness requirement as the system still checks at connect time to be sure any other sockets with the same local address and port do not have the same foreign address and port (if an association already exists, the error EADDRINUSE is returned).

Local address binding by the system is currently done somewhat haphazardly when a host is on multiple networks. Logically, one would expect the system to bind the local address associated with the network through which a peer was communicating. For instance, if the local host is connected to networks 46 and 10 and the foreign host is on network 32, and traffic from network 32 were arriving via network 10, the local address to be bound would be the host's address on network 10, not network 46. This unfortunately, is not always the case. For reasons too complicated to discuss here, the local address bound may be appear to be chosen at random. This property of local address binding will normally be invisible to users unless the foreign host does not understand how to reach the address selected*.

* For example, if network 46 were unknown to the host on network 32, and the local address were bound to that located on network 46, then even though a route between the two hosts existed through network 10, a connection would fail.

5.5. Broadcasting and datagram sockets

By using a datagram socket it is possible to send broadcast packets on many networks supported by the system (the network itself must support the notion of broadcasting; the system provides no broadcast simulation in software). Broadcast messages can place a high load on a network since they force every host on the network to service them. Consequently, the ability to send broadcast packets has been limited to the super user.

To send a broadcast message, an Internet datagram socket should be created:

```
s = socket(AF_INET, SOCK_DGRAM, 0);
```

and at least a port number should be bound to the socket:

```
sin.sin_family = AF_INET;
sin.sin_addr.s_addr = INADDR_ANY;
sin.sin_port = MYPORT;
bind(s, (char *)&sin, sizeof (sin));
```

Then the message should be addressed as:

```
dst.sin_family = AF_INET;
dst.sin_addr.s_addr = INADDR_ANY;
dst.sin_port = DESTPORT;
```

and, finally, a sendto call may be used:

```
sendto(s, buf, buflen, 0, &dst, sizeof (dst));
```

Received broadcast messages contain the senders address and port (datagram sockets are anchored before a message is allowed to go out).

5.6. Signals

Two new signals have been added to the system which may be used in conjunction with the interprocess communication facilities. The SIGURG signal is associated with the existence of an "urgent condition". The SIGIO signal is used with "interrupt driven i/o" (not presently implemented). SIGURG is currently supplied a process when out of band data is present at a socket. If multiple sockets have out of band data awaiting delivery, a select call may be used to determine those sockets with such data.

An old signal which is useful when constructing server processes is SIGCHLD. This signal is delivered to a process when any children processes have changed state. Normally servers use the signal to "reap" child processes after exiting. For example, the remote login server loop shown in Figure 2 may be augmented as follows:

```
int reaper();
 ...
sigset(SIGCHLD, reaper);
listen(f, 10);
for (;;) {
        int g, len = sizeof (from);

        g = accept(f, &from, &len, 0);
        if (g < 0) {
                if (errno != EINTR)
                        perror("rlogind: accept");
                continue;
        }
        ...
}
 ...
#include <wait.h>
reaper()
{
        union wait status;

        while (wait3(&status, WNOHANG, 0) > 0)
                ;
}
```

If the parent server process fails to reap its children, a large number of "zombie" processes may be created.

BERKELEY: COLLEGE OF ENGINEERING
ELECTRONICS RESEARCH LABORATORY
CORY HALL

July 18, 1983

TO: Professor Fateman

FR: Janet Henry

RE: XCS Deliverables

In order to extend funding of the current contract for the period 10/1/83 through 10/31/83 it is essential that all technical reports be on file at DARPA headquarters. In the attached documents I have noted the area you are responsible for, which we have not yet received a report. I would appreciate your supplying this as soon as possible so that we can get it through proper channels with no delay.

4.2BSD Networking Implementation Notes

Revised July, 1983

Samuel J. Leffler, William N. Joy, Robert S. Fabry

Computer Systems Research Group
Computer Science Division
Department of Electrical Engineering and Computer Science
University of California, Berkeley
Berkeley, CA 94720

(415) 642-7780

ABSTRACT

This report describes the internal structure of the networking facilities developed for the 4.2BSD version of the UNIX* operating system for the VAX†. These facilities are based on several central abstractions which structure the external (user) view of network communication as well as the internal (system) implementation.

The report documents the internal structure of the networking system. The "4.2BSD System Manual" provides a description of the user interface to the networking facilities.

* UNIX is a trademark of Bell Laboratories.
† DEC, VAX, DECnet, and UNIBUS are trademarks of Digital Equipment Corporation.

TABLE OF CONTENTS

1. Introduction

This report describes the internal structure of facilities added to the 4.2BSD version of the UNIX operating system for the VAX. The system facilities provide a uniform user interface to networking within UNIX. In addition, the implementation introduces a structure for network communications which may be used by system implementors in adding new networking facilities. The internal structure is not visible to the user, rather it is intended to aid implementors of communication protocols and network services by providing a framework which promotes code sharing and minimizes implementation effort.

The reader is expected to be familiar with the C programming language and system interface, as described in the *4.2BSD System Manual* [Joy82a]. Basic understanding of network communication concepts is assumed; where required any additional ideas are introduced.

The remainder of this document provides a description of the system internals, avoiding, when possible, those portions which are utilized only by the interprocess communication facilities.

2. Overview

If we consider the International Standards Organization's (ISO) Open System Interconnection (OSI) model of network communication [ISO81] [Zimmermann80], the networking facilities described here correspond to a portion of the session layer (layer 3) and all of the transport and network layers (layers 2 and 1, respectively).

The network layer provides possibly imperfect data transport services with minimal addressing structure. Addressing at this level is normally host to host, with implicit or explicit routing optionally supported by the communicating agents.

At the transport layer the notions of reliable transfer, data sequencing, flow control, and service addressing are normally included. Reliability is usually managed by explicit acknowledgement of data delivered. Failure to acknowledge a transfer results in retransmission of the data. Sequencing may be handled by tagging each message handed to the network layer by a *sequence number* and maintaining state at the endpoints of communication to utilize received sequence numbers in reordering data which arrives out of order.

The session layer facilities may provide forms of addressing which are mapped into formats required by the transport layer, service authentication and client authentication, etc. Various systems also provide services such as data encryption and address and protocol translation.

The following sections begin by describing some of the common data structures and utility routines, then examine the internal layering. The contents of each layer and its interface are considered. Certain of the interfaces are protocol implementation specific. For these cases examples have been drawn from the Internet [Cerf78] protocol family. Later sections cover routing issues, the design of the raw socket interface and other miscellaneous topics.

3. Goals

The networking system was designed with the goal of supporting multiple *protocol families* and addressing styles. This required information to be "hidden" in common data structures which could be manipulated by all the pieces of the system, but which required interpretation only by the protocols which "controlled" it. The system described here attempts to minimize the use of shared data structures to those kept by a suite of protocols (a *protocol family*), and those used for rendezvous between "synchronous" and "asynchronous" portions of the system (e.g. queues of data packets are filled at interrupt time and emptied based on user requests).

A major goal of the system was to provide a framework within which new protocols and hardware could be easily be supported. To this end, a great deal of effort has been extended to create utility routines which hide many of the more complex and/or hardware dependent chores of networking. Later sections describe the utility routines and the underlying data structures they manipulate.

4. Internal address representation

Common to all portions of the system are two data structures. These structures are used to represent addresses and various data objects. Addresses, internally are described by the *sockaddr* structure,

```
struct sockaddr {
        short       sa_family;      /* data format identifier */
        char        sa_data[14];    /* address */
};
```

All addresses belong to one or more *address families* which define their format and interpretation. The *sa_family* field indicates which address family the address belongs to, the *sa_data* field contains the actual data value. The size of the data field, 14 bytes, was selected based on a study of current address formats*.

* Later versions of the system support variable length addresses.

5. Memory management

A single mechanism is used for data storage: memory buffers, or *mbuf*'s. An mbuf is a structure of the form:

```
struct mbuf {
        struct  mbuf *m_next;   /* next buffer in chain */
        u_long  m_off;          /* offset of data */
        short   m_len;          /* amount of data in this mbuf */
        short   m_type;         /* mbuf type (accounting) */
        u_char  m_dat[MLEN];    /* data storage */
        struct  mbuf *m_act;    /* link in higher-level mbuf list */
};
```

The *m_next* field is used to chain mbufs together on linked lists, while the *m_act* field allows lists of mbufs to be accumulated. By convention, the mbufs common to a single object (for example, a packet) are chained together with the *m_next* field, while groups of objects are linked via the *m_act* field (possibly when in a queue).

Each mbuf has a small data area for storing information, *m_dat*. The *m_len* field indicates the amount of data, while the *m_off* field is an offset to the beginning of the data from the base of the mbuf. Thus, for example, the macro *mtod*, which converts a pointer to an mbuf to a pointer to the data stored in the mbuf, has the form

```
#define  mtod(x,t)          ((t)((int)(x) + (x)->m_off))
```

(note the *t* parameter, a C type cast, is used to cast the resultant pointer for proper assignment).

In addition to storing data directly in the mbuf's data area, data of page size may be also be stored in a separate area of memory. The mbuf utility routines maintain a pool of pages for this purpose and manipulate a private page map for such pages. The virtual addresses of these data pages precede those of mbufs, so when pages of data are separated from an mbuf, the mbuf data offset is a negative value. An array of reference counts on pages is also maintained so that copies of pages may be made without core to core copying (copies are created simply by duplicating the relevant page table entries in the data page map and incrementing the associated reference counts for the pages). Separate data pages are currently used only when copying data from a user process into the kernel, and when bringing data in at the hardware level. Routines which manipulate mbufs are not normally aware if data is stored directly in the mbuf data array, or if it is kept in separate pages.

The following utility routines are available for manipulating mbuf chains:

m = m_copy(m0, off, len);
: The *m_copy* routine create a copy of all, or part, of a list of the mbufs in *m0*. *Len* bytes of data, starting *off* bytes from the front of the chain, are copied. Where possible, reference counts on pages are used instead of core to core copies. The original mbuf chain must have at least *off* + *len* bytes of data. If *len* is specified as M_COPYALL, all the data present, offset as before, is copied.

m_cat(m, n);
: The mbuf chain, *n*, is appended to the end of *m*. Where possible, compaction is performed.

m_adj(m, diff);
: The mbuf chain, *m* is adjusted in size by *diff* bytes. If *diff* is non-negative, *diff* bytes are shaved off the front of the mbuf chain. If *diff* is negative, the alteration is performed from back to front. No space is reclaimed in this operation, alterations are accomplished by changing the *m_len* and *m_off* fields of mbufs.

m = m_pullup(m0, size);
: After a successful call to *m_pullup*, the mbuf at the head of the returned list, *m*, is

guaranteed to have at least *size* bytes of data in contiguous memory (allowing access via a pointer, obtained using the *mtod* macro). If the original data was less than *size* bytes long, *len* was greater than the size of an mbuf data area (112 bytes), or required resources were unavailable, *m* is 0 and the original mbuf chain is deallocated.

This routine is particularly useful when verifying packet header lengths on reception. For example, if a packet is received and only 8 of the necessary 16 bytes required for a valid packet header are present at the head of the list of mbufs representing the packet, the remaining 8 bytes may be "pulled up" with a single *m_pullup* call. If the call fails the invalid packet will have been discarded.

By insuring mbufs always reside on 128 byte boundaries it is possible to always locate the mbuf associated with a data area by masking off the low bits of the virtual address. This allows modules to store data structures in mbufs and pass them around without concern for locating the original mbuf when it comes time to free the structure. The *dtom* macro is used to convert a pointer into an mbuf's data area to a pointer to the mbuf,

```
#define    dtom(x)    ((struct mbuf *)((int)x & ~(MSIZE-1)))
```

Mbufs are used for dynamically allocated data structures such as sockets, as well as memory allocated for packets. Statistics are maintained on mbuf usage and can be viewed by users using the *netstat*(1) program.

6. Internal layering

The internal structure of the network system is divided into three layers. These layers correspond to the services provided by the socket abstraction, those provided by the communication protocols, and those provided by the hardware interfaces. The communication protocols are normally layered into two or more individual cooperating layers, though they are collectively viewed in the system as one layer providing services supportive of the appropriate socket abstraction.

The following sections describe the properties of each layer in the system and the interfaces each must conform to.

6.1. Socket layer

The socket layer deals with the interprocess communications facilities provided by the system. A socket is a bidirectional endpoint of communication which is "typed" by the semantics of communication it supports. The system calls described in the *4.2BSD System Manual* are used to manipulate sockets.

A socket consists of the following data structure:

```
struct socket {
        short       so_type;        /* generic type */
        short       so_options;     /* from socket call */
        short       so_linger;      /* time to linger while closing */
        short       so_state;       /* internal state flags */
        caddr_t     so_pcb;         /* protocol control block */
        struct      protosw *so_proto; /* protocol handle */
        struct      socket *so_head;  /* back pointer to accept socket */
        struct      socket *so_q0;    /* queue of partial connections */
        short       so_q0len;       /* partials on so_q0 */
        struct      socket *so_q;     /* queue of incoming connections */
        short       so_qlen;        /* number of connections on so_q */
        short       so_qlimit;      /* max number queued connections */
        struct      sockbuf so_snd; /* send queue */
        struct      sockbuf so_rcv; /* receive queue */
        short       so_timeo;       /* connection timeout */
        u_short     so_error;       /* error affecting connection */
        short       so_oobmark;     /* chars to oob mark */
        short       so_pgrp;        /* pgrp for signals */
};
```

Each socket contains two data queues, *so_rcv* and *so_snd*, and a pointer to routines which provide supporting services. The type of the socket, *so_type* is defined at socket creation time and used in selecting those services which are appropriate to support it. The supporting protocol is selected at socket creation time and recorded in the socket data structure for later use. Protocols are defined by a table of procedures, the *protosw* structure, which will be described in detail later. A pointer to a protocol specific data structure, the "protocol control block" is also present in the socket structure. Protocols control this data structure and it normally includes a back pointer to the parent socket structure(s) to allow easy lookup when returning information to a user (for example, placing an error number in the *so_error* field). The other entries in the socket structure are used in queueing connection requests, validating user requests, storing socket characteristics (e.g. options supplied at the time a socket is created), and maintaining a socket's state.

Processes "rendezvous at a socket" in many instances. For instance, when a process wishes to extract data from a socket's receive queue and it is empty, or lacks sufficient data to satisfy the request, the process blocks, supplying the address of the receive queue as an "wait channel' to be used in notification. When data arrives for the process and is placed in the

socket's queue, the blocked process is identified by the fact it is waiting "on the queue".

6.1.1. Socket state

A socket's state is defined from the following:

```
#define SS_NOFDREF              0x001   /* no file table ref any more */
#define SS_ISCONNECTED          0x002   /* socket connected to a peer */
#define SS_ISCONNECTING         0x004   /* in process of connecting to peer */
#define SS_ISDISCONNECTING      0x008   /* in process of disconnecting */
#define SS_CANTSENDMORE         0x010   /* can't send more data to peer */
#define SS_CANTRCVMORE          0x020   /* can't receive more data from peer */
#define SS_CONNAWAITING         0x040   /* connections awaiting acceptance */
#define SS_RCVATMARK            0x080   /* at mark on input */

#define SS_PRIV                 0x100   /* privileged */
#define SS_NBIO                 0x200   /* non-blocking ops */
#define SS_ASYNC                0x400   /* async i/o notify */
```

The state of a socket is manipulated both by the protocols and the user (through system calls). When a socket is created the state is defined based on the type of input/output the user wishes to perform. "Non-blocking" I/O implies a process should never be blocked to await resources. Instead, any call which would block returns prematurely with the error EWOULD-BLOCK (the service request may be partially fulfilled, e.g. a request for more data than is present).

If a process requested "asynchronous" notification of events related to the socket the SIGIO signal is posted to the process. An event is a change in the socket's state, examples of such occurances are: space becoming available in the send queue, new data available in the receive queue, connection establishment or disestablishment, etc.

A socket may be marked "priviledged" if it was created by the super-user. Only priviledged sockets may send broadcast packets, or bind addresses in priviledged portions of an address space.

6.1.2. Socket data queues

A socket's data queue contains a pointer to the data stored in the queue and other entries *related to the management of the data.* The following structure defines a data queue:

```
struct sockbuf {
        short       sb_cc;          /* actual chars in buffer */
        short       sb_hiwat;       /* max actual char count */
        short       sb_mbcnt;       /* chars of mbufs used */
        short       sb_mbmax;       /* max chars of mbufs to use */
        short       sb_lowat;       /* low water mark */
        short       sb_timeo;       /* timeout */
        struct      mbuf *sb_mb;    /* the mbuf chain */
        struct      proc *sb_sel;   /* process selecting read/write */
        short       sb_flags;       /* flags, see below */
};
```

Data is stored in a queue as a chain of mbufs. The actual count of characters as well as high and low water marks are used by the protocols in controlling the flow of data. The socket routines cooperate in implementing the flow control policy by blocking a process when it requests to send data and the high water mark has been reached, or when it requests to receive data and less than the low water mark is present (assuming non-blocking I/O has not been specified).

When a socket is created, the supporting protocol "reserves" space for the send and receive queues of the socket. The actual storage associated with a socket queue may fluctuate during a socket's lifetime, but is assumed this reservation will always allow a protocol to acquire enough memory to satisfy the high water marks.

The timeout and select values are manipulated by the socket routines in implementing various portions of the interprocess communications facilities and will not be described here.

A socket queue has a number of flags used in synchronizing access to the data and in acquiring resources;

```
#define SB_LOCK     0x01    /* lock on data queue (so_rcv only) */
#define SB_WANT     0x02    /* someone is waiting to lock */
#define SB_WAIT     0x04    /* someone is waiting for data/space */
#define SB_SEL      0x08    /* buffer is selected */
#define SB_COLL     0x10    /* collision selecting */
```

The last two flags are manipulated by the system in implementing the select mechanism.

6.1.3. Socket connection queueing

In dealing with connection oriented sockets (e.g. SOCK_STREAM) the two sides are considered distinct. One side is termed *active*, and generates connection requests. The other side is called *passive* and accepts connection requests.

From the passive side, a socket is created with the option SO_ACCEPTCONN specified, creating two queues of sockets: *so_q0* for connections in progress and *so_q* for connections already made and awaiting user acceptance. As a protocol is preparing incoming connections, it creates a socket structure queued on *so_q0* by calling the routine *sonewconn*(). When the connection is established, the socket structure is then transfered to *so_q*, making it available for an accept.

If an SO_ACCEPTCONN socket is closed with sockets on either *so_q0* or *so_q*, these sockets are dropped.

6.2. Protocol layer(s)

Protocols are described by a set of entry points and certain socket visible characteristics, some of which are used in deciding which socket type(s) they may support.

An entry in the "protocol switch" table exists for each protocol module configured into the system. It has the following form:

```
struct protosw {
        short       pr_type;            /* socket type used for */
        short       pr_family;          /* protocol family */
        short       pr_protocol;        /* protocol number */
        short       pr_flags;           /* socket visible attributes */
/* protocol-protocol hooks */
        int         (*pr_input)();      /* input to protocol (from below) */
        int         (*pr_output)();     /* output to protocol (from above) */
        int         (*pr_ctlinput)();   /* control input (from below) */
        int         (*pr_ctloutput)();  /* control output (from above) */
/* user-protocol hook */
        int         (*pr_usrreq)();     /* user request */
/* utility hooks */
        int         (*pr_init)();       /* initialization routine */
        int         (*pr_fasttimo)();   /* fast timeout (200ms) */
        int         (*pr_slowtimo)();   /* slow timeout (500ms) */
        int         (*pr_drain)();      /* flush any excess space possible */
};
```

A protocol is called through the *pr_init* entry before any other. Thereafter it is called every 200 milliseconds through the *pr_fasttimo* entry and every 500 milliseconds through the *pr_slowtimo* for timer based actions. The system will call the *pr_drain* entry if it is low on space and this should throw away any non-critical data.

Protocols pass data between themselves as chains of mbufs using the *pr_input* and *pr_output* routines. *Pr_input* passes data up (towards the user) and *pr_output* passes it down (towards the network); control information passes up and down on *pr_ctlinput* and *pr_ctloutput*. The protocol is responsible for the space occupied by any the arguments to these entries and must dispose of it.

The *pr_userreq* routine interfaces protocols to the socket code and is described below.

The *pr_flags* field is constructed from the following values:

```
#define PR_ATOMIC           0x01    /* exchange atomic messages only */
#define PR_ADDR             0x02    /* addresses given with messages */
#define PR_CONNREQUIRED     0x04    /* connection required by protocol */
#define PR_WANTRCVD         0x08    /* want PRU_RCVD calls */
#define PR_RIGHTS           0x10    /* passes capabilities */
```

Protocols which are connection-based specify the PR_CONNREQUIRED flag so that the socket routines will never attempt to send data before a connection has been established. If the PR_WANTRCVD flag is set, the socket routines will notfiy the protocol when the user has removed data from the socket's receive queue. This allows the protocol to implement acknowledgement on user receipt, and also update windowing information based on the amount of space available in the receive queue. The PR_ADDR field indicates any data placed in the socket's receive queue will be preceded by the address of the sender. The PR_ATOMIC flag specifies each *user* request to send data must be performed in a single *protocol* send request; it is the protocol's responsibility to maintain record boundaries on data to be sent. The PR_RIGHTS flag indicates the protocol supports the passing of capabilities; this is currently used only the protocols in the UNIX protocol family.

When a socket is created, the socket routines scan the protocol table looking for an appropriate protocol to support the type of socket being created. The *pr_type* field contains one of the possible socket types (e.g. SOCK_STREAM), while the *pr_family* field indicates which protocol family the protocol belongs to. The *pr_protocol* field contains the protocol number of the protocol, normally a well known value.

6.3. Network-interface layer

Each network-interface configured into a system defines a path through which packets may be sent and received. Normally a hardware device is associated with this interface, though there is no requirement for this (for example, all systems have a software "loopback" interface used for debugging and performance analysis). In addition to manipulating the hardware device, an *interface module* is responsible for encapsulation and deencapsulation of any low level header information required to deliver a message to it's destination. The selection of which interface to use in delivering packets is a routing decision carried out at a higher level than the network-interface layer. Each interface normally identifies itself at boot time to the routing module so that it may be selected for packet delivery.

An interface is defined by the following structure,

```
struct ifnet {
        char    *if_name;           /* name, e.g. "en" or "lo" */
        short   if_unit;            /* sub-unit for lower level driver */
        short   if_mtu;             /* maximum transmission unit */
        int     if_net;             /* network number of interface */
        short   if_flags;           /* up/down, broadcast, etc. */
        short   if_timer;           /* time 'til if_watchdog called */
        int     if_host[2];         /* local net host number */
        struct  sockaddr if_addr;   /* address of interface */
        union {
                struct  sockaddr ifu_broadaddr;
                struct  sockaddr ifu_dstaddr;
        } if_ifu;
        struct  ifqueue if_snd;     /* output queue */
        int     (*if_init)();       /* init routine */
        int     (*if_output)();     /* output routine */
        int     (*if_ioctl)();      /* ioctl routine */
        int     (*if_reset)();      /* bus reset routine */
        int     (*if_watchdog)();   /* timer routine */
        int     if_ipackets;        /* packets received on interface */
        int     if_ierrors;         /* input errors on interface */
        int     if_opackets;        /* packets sent on interface */
        int     if_oerrors;         /* output errors on interface */
        int     if_collisions;      /* collisions on csma interfaces */
        struct  ifnet *if_next;
};
```

Each interface has a send queue and routines used for initialization, *if_init*, and output, *if_output*. If the interface resides on a system bus, the routine *if_reset* will be called after a bus reset has been performed. An interface may also specify a timer routine, *if_watchdog*, which should be called every *if_timer* seconds (if non-zero).

The state of an interface and certain characteristics are stored in the *if_flags* field. The following values are possible:

```
#define IFF_UP              0x1     /* interface is up */
#define IFF_BROADCAST       0x2     /* broadcast address valid */
#define IFF_DEBUG           0x4     /* turn on debugging */
#define IFF_ROUTE           0x8     /* routing entry installed */
#define IFF_POINTOPOINT     0x10    /* interface is point-to-point link */
#define IFF_NOTRAILERS      0x20    /* avoid use of trailers */
#define IFF_RUNNING         0x40    /* resources allocated */
```

If the interface is connected to a network which supports transmission of *broadcast* packets, the

IFF_BROADCAST flag will be set and the *if_broadaddr* field will contain the address to be used in sending or accepting a broadcast packet. If the interface is associated with a point to point hardware link (for example, a DEC DMR-11), the IFF_POINTOPOINT flag will be set and *if_dstaddr* will contain the address of the host on the other side of the connection. These addresses and the local address of the interface, *if_addr*, are used in filtering incoming packets. The interface sets IFF_RUNNING after it has allocated system resources and posted an initial read on the device it manages. This state bit is used to avoid multiple allocation requests when an interface's address is changed. The IFF_NOTRAILERS flag indicates the interface should refrain from using a *trailer* encapsulation on outgoing packets; *trailer* protocols are described in section 14.

The information stored in an *ifnet* structure for point to point communication devices is not currently used by the system internally. Rather, it is used by the user level routing process in determining host network connections and in initially devising routes (refer to chapter 10 for more information).

Various statistics are also stored in the interface structure. These may be viewed by users using the *netstat*(1) program.

The interface address and flags may be set with the SIOCSIFADDR and SIOCSIFFLAGS ioctls. SIOCSIFADDR is used to initially define each interface's address; SIOGSIFFLAGS can be used to mark an interface down and perform site-specific configuration.

6.3.1. UNIBUS interfaces

All hardware related interfaces currently reside on the UNIBUS. Consequently a common set of utility routines for dealing with the UNIBUS has been developed. Each UNIBUS interface utilizes a structure of the following form:

```
struct  ifuba {
        short       ifu_uban;               /* uba number */
        short       ifu_hlen;               /* local net header length */
        struct      uba_regs *ifu_uba;      /* uba regs, in vm */
        struct ifrw {
                    caddr_t     ifrw_addr;  /* virt addr of header */
                    int         ifrw_bdp;   /* unibus bdp */
                    int         ifrw_info;  /* value from ubaalloc */
                    int         ifrw_proto; /* map register prototype */
                    struct      pte *ifrw_mr;/* base of map registers */
        } ifu_r, ifu_w;
        struct      pte ifu_wmap[IF_MAXNUBAMR];/* base pages for output */
        short       ifu_xswapd;             /* mask of clusters swapped */
        short       ifu_flags;              /* used during uballoc's */
        struct      mbuf *ifu_xtofree;      /* pages being dma'd out */
};
```

The *if_uba* structure describes UNIBUS resources held by an interface. IF_NUBAMR map registers are held for datagram data, starting at *ifr_mr*. UNIBUS map register *ifr_mr*[−1] maps the local network header ending on a page boundary. UNIBUS data paths are reserved for read and for write, given by *ifr_bdp*. The prototype of the map registers for read and for write is saved in *ifr_proto*.

When write transfers are not full pages on page boundaries the data is just copied into the pages mapped on the UNIBUS and the transfer is started. If a write transfer is of a (1024 byte) page size and on a page boundary, UNIBUS page table entries are swapped to reference the pages, and then the initial pages are remapped from *ifu_wmap* when the transfer completes.

When read transfers give whole pages of data to be input, page frames are allocated from a network page list and traded with the pages already containing the data, mapping the allocated

pages to replace the input pages for the next UNIBUS data input.

The following utility routines are available for use in writing network interface drivers, all use the *ifuba* structure described above.

if_ubainit(ifu, uban, hlen, nmr);

if_ubainit allocates resources on UNIBUS adaptor *uban* and stores the resultant information in the *ifuba* structure pointed to by *ifu*. It is called only at boot time or after a UNIBUS reset. Two data paths (buffered or unbuffered, depending on the *ifu_flags* field) are allocated, one for reading and one for writing. The *nmr* parameter indicates the number of UNIBUS mapping registers required to map a maximal sized packet onto the UNIBUS, while *hlen* specifies the size of a local network header, if any, which should be mapped separately from the data (see the description of trailer protocols in chapter 14). Sufficient UNIBUS mapping registers and pages of memory are allocated to initialize the input data path for an initial read. For the output data path, mapping registers and pages of memory are also allocated and mapped onto the UNIBUS. The pages associated with the output data path are held in reserve in the event a write requires copying non-page-aligned data (see *if_wubaput* below). If *if_ubainit* is called with resources already allocated, they will be used instead of allocating new ones (this normally occurs after a UNIBUS reset). A 1 is returned when allocation and initialization is successful, 0 otherwise.

m = if_rubaget(ifu, totlen, off0);

if_rubaget pulls read data off an interface. *totlen* specifies the length of data to be obtained, not counting the local network header. If *off0* is non-zero, it indicates a byte offset to a trailing local network header which should be copied into a separate mbuf and prepended to the front of the resultant mbuf chain. When page sized units of data are present and are page-aligned, the previously mapped data pages are remapped into the mbufs and swapped with fresh pages; thus avoiding any copying. A 0 return value indicates a failure to allocate resources.

if_wubaput(ifu, m);

if_wubaput maps a chain of mbufs onto a network interface in preparation for output. The chain includes any local network header, which is copied so that it resides in the mapped and aligned I/O space. Any other mbufs which contained non page sized data portions are also copied to the I/O space. Pages mapped from a previous output operation (no longer needed) are unmapped and returned to the network page pool.

7. Socket/protocol interface

The interface between the socket routines and the communication protocols is through the *pr_usrreq* routine defined in the protocol switch table. The following requests to a protocol module are possible:

```
#define PRU_ATTACH      0    /* attach protocol */
#define PRU_DETACH      1    /* detach protocol */
#define PRU_BIND        2    /* bind socket to address */
#define PRU_LISTEN      3    /* listen for connection */
#define PRU_CONNECT     4    /* establish connection to peer */
#define PRU_ACCEPT      5    /* accept connection from peer */
#define PRU_DISCONNECT  6    /* disconnect from peer */
#define PRU_SHUTDOWN    7    /* won't send any more data */
#define PRU_RCVD        8    /* have taken data; more room now */
#define PRU_SEND        9    /* send this data */
#define PRU_ABORT       10   /* abort (fast DISCONNECT, DETATCH) */
#define PRU_CONTROL     11   /* control operations on protocol */
#define PRU_SENSE       12   /* return status into m */
#define PRU_RCVOOB      13   /* retrieve out of band data */
#define PRU_SENDOOB     14   /* send out of band data */
#define PRU_SOCKADDR    15   /* fetch socket's address */
#define PRU_PEERADDR    16   /* fetch peer's address */
#define PRU_CONNECT2    17   /* connect two sockets */
/* begin for protocols internal use */
#define PRU_FASTTIMO    18   /* 200ms timeout */
#define PRU_SLOWTIMO    19   /* 500ms timeout */
#define PRU_PROTORCV    20   /* receive from below */
#define PRU_PROTOSEND   21   /* send to below */
```

A call on the user request routine is of the form,

```
error = (*protosw[].pr_usrreq)(up, req, m, addr, rights);
int error; struct socket *up; int req; struct mbuf *m, *rights; caddr_t addr;
```

The mbuf chain, *m*, and the address are optional parameters. The *rights* parameter is an optional pointer to an mbuf chain containing user specified capabilities (see the *sendmsg* and *recvmsg* system calls). The protocol is responsible for disposal of both mbuf chains. A non-zero return value gives a UNIX error number which should be passed to higher level software. The following paragraphs describe each of the requests possible.

PRU_ATTACH
When a protocol is bound to a socket (with the *socreate* system call) the protocol module is called with this request. It is the responsibility of the protocol module to allocate any resources necessary. The "attach" request will always precede any of the other requests, and should not occur more than once.

PRU_DETACH
This is the antithesis of the attach request, and is used at the time a socket is deleted. The protocol module may deallocate any resources assigned to the socket.

PRU_BIND
When a socket is initially created it has no address bound to it. This request indicates an address should be bound to an existing socket. The protocol module must verify the requested address is valid and available for use.

PRU_LISTEN
The "listen" request indicates the user wishes to listen for incoming connection requests on the associated socket. The protocol module should perform any state changes needed to carry out this request (if possible). A "listen" request always precedes a request to

accept a connection.

PRU_CONNECT
The "connect" request indicates the user wants to a establish an association. The *addr* parameter supplied describes the peer to be connected to. The effect of a connect request may vary depending on the protocol. Virtual circuit protocols, such as TCP [Postel80b], use this request to initiate establishment of a TCP connection. Datagram protocols, such as UDP [Postel79], simply record the peer's address in a private data structure and use it to tag all outgoing packets. There are no restrictions on how many times a connect request may be used after an attach. If a protocol supports the notion of *multi-casting*, it is possible to use multiple connects to establish a multi-cast group. Alternatively, an association may be broken by a PRU_DISCONNECT request, and a new association created with a subsequent connect request; all without destroying and creating a new socket.

PRU_ACCEPT
Following a successful PRU_LISTEN request and the arrival of one or more connections, this request is made to indicate the user has accepted the first connection on the queue of pending connections. The protocol module should fill in the supplied address buffer with the address of the connected party.

PRU_DISCONNECT
Eliminate an association created with a PRU_CONNECT request.

PRU_SHUTDOWN
This call is used to indicate no more data will be sent and/or received (the *addr* parameter indicates the direction of the shutdown, as encoded in the *soshutdown* system call). The protocol may, at its discretion, deallocate any data structures related to the shutdown.

PRU_RCVD
This request is made only if the protocol entry in the protocol switch table includes the PR_WANTRCVD flag. When a user removes data from the receive queue this request will be sent to the protocol module. It may be used to trigger acknowledgements, refresh windowing information, initiate data transfer, etc.

PRU_SEND
Each user request to send data is translated into one or more PRU_SEND requests (a protocol may indicate a single user send request must be translated into a single PRU_SEND request by specifying the PR_ATOMIC flag in its protocol description). The data to be sent is presented to the protocol as a list of mbufs and an address is, optionally, supplied in the *addr* parameter. The protocol is responsible for preserving the data in the socket's send queue if it is not able to send it immediately, or if it may need it at some later time (e.g. for retransmission).

PRU_ABORT
This request indicates an abnormal termination of service. The protocol should delete any existing association(s).

PRU_CONTROL
The "control" request is generated when a user performs a UNIX *ioctl* system call on a socket (and the ioctl is not intercepted by the socket routines). It allows protocol-specific operations to be provided outside the scope of the common socket interface. The *addr* parameter contains a pointer to a static kernel data area where relevant information may be obtained or returned. The *m* parameter contains the actual *ioctl* request code (note the non-standard calling convention).

PRU_SENSE
The "sense" request is generated when the user makes an *fstat* system call on a socket; it requests status of the associated socket. There currently is no common format for the status returned. Information which might be returned includes per-connection statistics, protocol state, resources currently in use by the connection, the optimal transfer size for the connection (based on windowing information and maximum packet size). The *addr*

parameter contains a pointer to a static kernel data area where the status buffer should be placed.

PRU_RCVOOB
: Any "out-of-band" data presently available is to be returned. An mbuf is passed in to the protocol module and the protocol should either place data in the mbuf or attach new mbufs to the one supplied if there is insufficient space in the single mbuf.

PRU_SENDOOB
: Like PRU_SEND, but for out-of-band data.

PRU_SOCKADDR
: The local address of the socket is returned, if any is currently bound to the it. The address format (protocol specific) is returned in the *addr* parameter.

PRU_PEERADDR
: The address of the peer to which the socket is connected is returned. The socket must be in a SS_ISCONNECTED state for this request to be made to the protocol. The address format (protocol specific) is returned in the *addr* parameter.

PRU_CONNECT2
: The protocol module is supplied two sockets and requested to establish a connection between the two without binding any addresses, if possible. This call is used in implementing the system call.

The following requests are used internally by the protocol modules and are never generated by the socket routines. In certain instances, they are handed to the *pr_usrreq* routine solely for convenience in tracing a protocol's operation (e.g. PRU_SLOWTIMO).

PRU_FASTTIMO
: A "fast timeout" has occured. This request is made when a timeout occurs in the protocol's *pr_fastimo* routine. The *addr* parameter indicates which timer expired.

PRU_SLOWTIMO
: A "slow timeout" has occured. This request is made when a timeout occurs in the protocol's *pr_slowtimo* routine. The *addr* parameter indicates which timer expired.

PRU_PROTORCV
: This request is used in the protocol-protocol interface, not by the routines. It requests reception of data destined for the protocol and not the user. No protocols currently use this facility.

PRU_PROTOSEND
: This request allows a protocol to send data destined for another protocol module, not a user. The details of how data is marked "addressed to protocol" instead of "addressed to user" are left to the protocol modules. No protocols currently use this facility.

8. Protocol/protocol interface

The interface between protocol modules is through the *pr_usrreq*, *pr_input*, *pr_output*, *pr_ctlinput*, and *pr_ctloutput* routines. The calling conventions for all but the *pr_usrreq* routine are expected to be specific to the protocol modules and are not guaranteed to be consistent across protocol families. We will examine the conventions used for some of the Internet protocols in this section as an example.

8.1. pr_output

The Internet protocol UDP uses the convention,

```
error = udp_output(inp, m);
int error; struct inpcb *inp; struct mbuf *m;
```

where the *inp*, "*in*ternet *p*rotocol *c*ontrol *b*lock", passed between modules conveys per connection state information, and the mbuf chain contains the data to be sent. UDP performs consistency checks, appends its header, calculates a checksum, etc. before passing the packet on to the IP module:

```
error = ip_output(m, opt, ro, allowbroadcast);
int error; struct mbuf *m, *opt; struct route *ro; int allowbroadcast;
```

The call to IP's output routine is more complicated than that for UDP, as befits the additional work the IP module must do. The *m* parameter is the data to be sent, and the *opt* parameter is an optional list of IP options which should be placed in the IP packet header. The *ro* parameter is is used in making routing decisions (and passing them back to the caller). The final parameter, *allowbroadcast* is a flag indicating if the user is allowed to transmit a broadcast packet. This may be inconsequential if the underlying hardware does not support the notion of broadcasting.

All output routines return 0 on success and a UNIX error number if a failure occured which could be immediately detected (no buffer space available, no route to destination, etc.).

8.2. pr_input

Both UDP and TCP use the following calling convention,

```
(void) (*protosw[].pr_input)(m);
struct mbuf *m;
```

Each mbuf list passed is a single packet to be processed by the protocol module.

The IP input routine is a VAX software interrupt level routine, and so is not called with any parameters. It instead communicates with network interfaces through a queue, *ipintrq*, which is identical in structure to the queues used by the network interfaces for storing packets awaiting transmission.

8.3. pr_ctlinput

This routine is used to convey "control" information to a protocol module (i.e. information which might be passed to the user, but is not data). This routine, and the *pr_ctloutput* routine, have not been extensively developed, and thus suffer from a "clumsiness" that can only be improved as more demands are placed on it.

The common calling convention for this routine is,

```
(void) (*protosw[].pr_ctlinput)(req, info);
int req; caddr_t info;
```

The *req* parameter is one of the following,

```
#define PRC_IFDOWN              0   /* interface transition */
#define PRC_ROUTEDEAD           1   /* select new route if possible */
#define PRC_QUENCH              4   /* some said to slow down */
#define PRC_HOSTDEAD            6   /* normally from IMP */
#define PRC_HOSTUNREACH         7   /* ditto */
#define PRC_UNREACH_NET         8   /* no route to network */
#define PRC_UNREACH_HOST        9   /* no route to host */
#define PRC_UNREACH_PROTOCOL    10  /* dst says bad protocol */
#define PRC_UNREACH_PORT        11  /* bad port # */
#define PRC_MSGSIZE             12  /* message size forced drop */
#define PRC_REDIRECT_NET        13  /* net routing redirect */
#define PRC_REDIRECT_HOST       14  /* host routing redirect */
#define PRC_TIMXCEED_INTRANS    17  /* packet lifetime expired in transit */
#define PRC_TIMXCEED_REASS      18  /* lifetime expired on reass q */
#define PRC_PARAMPROB           19  /* header incorrect */
```

while the *info* parameter is a "catchall" value which is request dependent. Many of the requests have obviously been derived from ICMP (the Internet Control Message Protocol), and from error messages defined in the 1822 host/IMP convention [BBN78]. Mapping tables exist to convert control requests to UNIX error codes which are delivered to a user.

8.4. pr_ctloutput

This routine is not currently used by any protocol modules.

9. Protocol/network-interface interface

The lowest layer in the set of protocols which comprise a protocol family must interface itself to one or more network interfaces in order to transmit and receive packets. It is assumed that any routing decisions have been made before handing a packet to a network interface, in fact this is absolutely necessary in order to locate any interface at all (unless, of course, one uses a single "hardwired" interface). There are two cases to be concerned with, transmission of a packet, and receipt of a packet; each will be considered separately.

9.1. Packet transmission

Assuming a protocol has a handle on an interface, *ifp*, a (struct ifnet *), it transmits a fully formatted packet with the following call,

```
error = (*ifp->if_output)(ifp, m, dst)
int error; struct ifnet *ifp; struct mbuf *m; struct sockaddr *dst;
```

The output routine for the network interface transmits the packet *m* to the *dst* address, or returns an error indication (a UNIX error number). In reality transmission may not be immediate, or successful; normally the output routine simply queues the packet on its send queue and primes an interrupt driven routine to actually transmit the packet. For unreliable mediums, such as the Ethernet, "successful" transmission simply means the packet has been placed on the cable without a collision. On the other hand, an 1822 interface guarantees proper delivery or an error indication for each message transmitted. The model employed in the networking system attaches no promises of delivery to the packets handed to a network interface, and thus corresponds more closely to the Ethernet. Errors returned by the output routine are normally trivial in nature (no buffer space, address format not handled, etc.).

9.2. Packet reception

Each protocol family must have one or more "lowest level" protocols. These protocols deal with internetwork addressing and are responsible for the delivery of incoming packets to the proper protocol processing modules. In the PUP model [Boggs78] these protocols are termed Level 1 protocols, in the ISO model, network layer protocols. In our system each such protocol module has an input packet queue assigned to it. Incoming packets received by a network interface are queued up for the protocol module and a VAX software interrupt is posted to initiate processing.

Three macros are available for queueing and dequeueing packets,

IF_ENQUEUE(ifq, m)
: This places the packet *m* at the tail of the queue *ifq*.

IF_DEQUEUE(ifq, m)
: This places a pointer to the packet at the head of queue *ifq* in *m*. A zero value will be returned in *m* if the queue is empty.

IF_PREPEND(ifq, m)
: This places the packet *m* at the head of the queue *ifq*.

Each queue has a maximum length associated with it as a simple form of congestion control. The macro IF_QFULL(ifq) returns 1 if the queue is filled, in which case the macro IF_DROP(ifq) should be used to bump a count of the number of packets dropped and the offending packet dropped. For example, the following code fragment is commonly found in a network interface's input routine,

```
if (IF_QFULL(inq)) {
        IF_DROP(inq);
        m_freem(m);
} else
        IF_ENQUEUE(inq, m);
```

10. Gateways and routing issues

The system has been designed with the expectation that it will be used in an internetwork environment. The "canonical" environment was envisioned to be a collection of local area networks connected at one or more points through hosts with multiple network interfaces (one on each local area network), and possibly a connection to a long haul network (for example, the ARPANET). In such an environment, issues of gatewaying and packet routing become very important. Certain of these issues, such as congestion control, have been handled in a simplistic manner or specifically not addressed. Instead, where possible, the network system attempts to provide simple mechanisms upon which more involved policies may be implemented. As some of these problems become better understood, the solutions developed will be incorporated into the system.

This section will describe the facilities provided for packet routing. The simplistic mechanisms provided for congestion control are described in chapter 12.

10.1. Routing tables

The network system maintains a set of routing tables for selecting a network interface to use in delivering a packet to its destination. These tables are of the form:

```
struct rtentry {
        u_long      rt_hash;        /* hash key for lookups */
        struct      sockaddr rt_dst;    /* destination net or host */
        struct      sockaddr rt_gateway;/* forwarding agent */
        short       rt_flags;       /* see below */
        short       rt_refcnt;      /* no. of references to structure */
        u_long      rt_use;         /* packets sent using route */
        struct      ifnet *rt_ifp;      /* interface to give packet to */
};
```

The routing information is organized in two separate tables, one for routes to a host and one for routes to a network. The distinction between hosts and networks is necessary so that a single mechanism may be used for both broadcast and multi-drop type networks, and also for networks built from point-to-point links (e.g DECnet [DEC80]).

Each table is organized as a hashed set of linked lists. Two 32-bit hash values are calculated by routines defined for each address family; one based on the destination being a host, and one assuming the target is the network portion of the address. Each hash value is used to locate a hash chain to search (by taking the value modulo the hash table size) and the entire 32-bit value is then used as a key in scanning the list of routes. Lookups are applied first to the routing table for hosts, then to the routing table for networks. If both lookups fail, a final lookup is made for a "wildcard" route (by convention, network 0). By doing this, routes to a specific host on a network may be present as well as routes to the network. This also allows a "fall back" network route to be defined to an "smart" gateway which may then perform more intelligent routing.

Each routing table entry contains a destination (who's at the other end of the route), a gateway to send the packet to, and various flags which indicate the route's status and type (host or network). A count of the number of packets sent using the route is kept for use in deciding between multiple routes to the same destination (see below), and a count of "held references" to the dynamically allocated structure is maintained to insure memory reclamation occurs only when the route is not in use. Finally a pointer to the a network interface is kept; packets sent using the route should be handed to this interface.

Routes are typed in two ways: either as host or network, and as "direct" or "indirect". The host/network distinction determines how to compare the *rt_dst* field during lookup. If the route is to a network, only a packet's destination network is compared to the *rt_dst* entry stored in the table. If the route is to a host, the addresses must match bit for bit.

The distinction between "direct" and "indirect" routes indicates whether the destination is directly connected to the source. This is needed when performing local network encapsulation. If a packet is destined for a peer at a host or network which is not directly connected to the source, the internetwork packet header will indicate the address of the eventual destination, while the local network header will indicate the address of the intervening gateway. Should the destination be directly connected, these addresses are likely to be identical, or a mapping between the two exists. The RTF_GATEWAY flag indicates the route is to an "indirect" gateway agent and the local network header should be filled in from the *rt_gateway* field instead of *rt_dst*, or from the internetwork destination address.

It is assumed multiple routes to the same destination will not be present unless they are deemed *equal* in cost (the current routing policy process never installs multiple routes to the same destination). However, should multiple routes to the same destination exist, a request for a route will return the "least used" route based on the total number of packets sent along this route. This can result in a "ping-pong" effect (alternate packets taking alternate routes), unless protocols "hold onto" routes until they no longer find them useful; either because the destination has changed, or because the route is lossy.

Routing redirect control messages are used to dynamically modify existing routing table entries as well as dynamically create new routing table entries. On hosts where exhaustive routing information is too expensive to maintain (e.g. work stations), the combination of wildcard routing entries and routing redirect messages can be used to provide a simple routing management scheme without the use of a higher level policy process. Statistics are kept by the routing table routines on the use of routing redirect messages and their affect on the routing tables. These statistics may be viewed using

Status information other than routing redirect control messages may be used in the future, but at present they are ignored. Likewise, more intelligent "metrics" may be used to describe routes in the future, possibly based on bandwidth and monetary costs.

10.2. Routing table interface

A protocol accesses the routing tables through three routines, one to allocate a route, one to free a route, and one to process a routing redirect control message. The routine *rtalloc* performs route allocation; it is called with a pointer to the following structure,

```
struct route {
        struct      rtentry *ro_rt;
        struct      sockaddr ro_dst;
};
```

The route returned is assumed "held" by the caller until disposed of with an *rtfree* call. Protocols which implement virtual circuits, such as TCP, hold onto routes for the duration of the circuit's lifetime, while connection-less protocols, such as UDP, currently allocate and free routes on each transmission.

The routine *rtredirect* is called to process a routing redirect control message. It is called with a destination address and the new gateway to that destination. If a non-wildcard route exists to the destination, the gateway entry in the route is modified to point at the new gateway supplied. Otherwise, a new routing table entry is inserted reflecting the information supplied. Routes to interfaces and routes to gateways which are not directly accesible from the host are ignored.

10.3. User level routing policies

Routing policies implemented in user processes manipulate the kernel routing tables through two *ioctl* calls. The commands SIOCADDRT and SIOCDELRT add and delete routing entries, respectively; the tables are read through the /dev/kmem device. The decision to place policy decisions in a user process implies routing table updates may lag a bit behind the identification of new routes, or the failure of existing routes, but this period of instability is

normally very small with proper implementation of the routing process. Advisory information, such as ICMP error messages and IMP diagnostic messages, may be read from raw sockets (described in the next section).

One routing policy process has already been implemented. The system standard "routing daemon" uses a variant of the Xerox NS Routing Information Protocol [Xerox82] to maintain up to date routing tables in our local environment. Interaction with other existing routing protocols, such as the Internet GGP (Gateway-Gateway Protocol), may be accomplished using a similar process.

11. Raw sockets

A raw socket is a mechanism which allows users direct access to a lower level protocol. Raw sockets are intended for knowledgeable processes which wish to take advantage of some protocol feature not directly accessible through the normal interface, or for the development of new protocols built atop existing lower level protocols. For example, a new version of TCP might be developed at the user level by utilizing a raw IP socket for delivery of packets. The raw IP socket interface attempts to provide an identical interface to the one a protocol would have if it were resident in the kernel.

The raw socket support is built around a generic raw socket interface, and (possibly) augmented by protocol-specific processing routines. This section will describe the core of the raw socket interface.

11.1. Control blocks

Every raw socket has a protocol control block of the following form,

```
struct rawcb {
        struct      rawcb *rcb_next;        /* doubly linked list */
        struct      rawcb *rcb_prev;
        struct      socket *rcb_socket;     /* back pointer to socket */
        struct      sockaddr rcb_faddr;     /* destination address */
        struct      sockaddr rcb_laddr;     /* socket's address */
        caddr_t     rcb_pcb;                /* protocol specific stuff */
        short       rcb_flags;
};
```

All the control blocks are kept on a doubly linked list for performing lookups during packet dispatch. Associations may be recorded in the control block and used by the output routine in preparing packets for transmission. The addresses are also used to filter packets on input; this will be described in more detail shortly. If any protocol specific information is required, it may be attached to the control block using the *rcb_pcb* field.

A raw socket interface is datagram oriented. That is, each send or receive on the socket requires a destination address. This address may be supplied by the user or stored in the control block and automatically installed in the outgoing packet by the output routine. Since it is not possible to determine whether an address is present or not in the control block, two flags, RAW_LADDR and RAW_FADDR, indicate if a local and foreign address are present. Another flag, RAW_DONTROUTE, indicates if routing should be performed on outgoing packets. If it is, a route is expected to be allocated for each "new" destination address. That is, the first time a packet is transmitted a route is determined, and thereafter each time the destination address stored in *rcb_route* differs from *rcb_faddr*, or *rcb_route.ro_rt* is zero, the old route is discarded and a new one allocated.

11.2. Input processing

Input packets are "assigned" to raw sockets based on a simple pattern matching scheme. Each network interface or protocol gives packets to the raw input routine with the call:

```
raw_input(m, proto, src, dst)
struct mbuf *m; struct sockproto *proto, struct sockaddr *src, *dst;
```

The data packet then has a generic header prepended to it of the form

```
struct raw_header {
        struct      sockproto raw_proto;
        struct      sockaddr raw_dst;
        struct      sockaddr raw_src;
};
```

and it is placed in a packet queue for the "raw input protocol" module. Packets taken from this queue are copied into any raw sockets that match the header according to the following rules,

1) The protocol family of the socket and header agree.
2) If the protocol number in the socket is non-zero, then it agrees with that found in the packet header.
3) If a local address is defined for the socket, the address format of the local address is the same as the destination address's and the two addresses agree bit for bit.
4) The rules of 3) are applied to the socket's foreign address and the packet's source address.

A basic assumption is that addresses present in the control block and packet header (as constructed by the network interface and any raw input protocol module) are in a canonical form which may be "block compared".

11.3. Output processing

On output the raw *pr_usrreq* routine passes the packet and raw control block to the raw protocol output routine for any processing required before it is delivered to the appropriate network interface. The output routine is normally the only code required to implement a raw socket interface.

12. Buffering and congestion control

One of the major factors in the performance of a protocol is the buffering policy used. Lack of a proper buffering policy can force packets to be dropped, cause falsified windowing information to be emitted by protocols, fragment host memory, degrade the overall host performance, etc. Due to problems such as these, most systems allocate a fixed pool of memory to the networking system and impose a policy optimized for "normal" network operation.

The networking system developed for UNIX is little different in this respect. At boot time a fixed amount of memory is allocated by the networking system. At later times more system memory may be requested as the need arises, but at no time is memory ever returned to the system. It is possible to garbage collect memory from the network, but difficult. In order to perform this garbage collection properly, some portion of the network will have to be "turned off" as data structures are updated. The interval over which this occurs must kept small compared to the average inter-packet arrival time, or too much traffic may be lost, impacting other hosts on the network, as well as increasing load on the interconnecting mediums. In our environment we have not experienced a need for such compaction, and thus have left the problem unresolved.

The mbuf structure was introduced in chapter 5. In this section a brief description will be given of the allocation mechanisms, and policies used by the protocols in performing connection level buffering.

12.1. Memory management

The basic memory allocation routines place no restrictions on the amount of space which may be allocated. Any request made is filled until the system memory allocator starts refusing to allocate additional memory. When the current quota of memory is insufficient to satisfy an mbuf allocation request, the allocator requests enough new pages from the system to satisfy the current request only. All memory owned by the network is described by a private page table used in remapping pages to be logically contiguous as the need arises. In addition, an array of reference counts parallels the page table and is used when multiple copies of a page are present.

Mbufs are 128 byte structures, 8 fitting in a 1Kbyte page of memory. When data is placed in mbufs, if possible, it is copied or remapped into logically contiguous pages of memory from the network page pool. Data smaller than the size of a page is copied into one or more 112 byte mbuf data areas.

12.2. Protocol buffering policies

Protocols reserve fixed amounts of buffering for send and receive queues at socket creation time. These amounts define the high and low water marks used by the socket routines in deciding when to block and unblock a process. The reservation of space does not currently result in any action by the memory management routines, though it is clear if one imposed an upper bound on the total amount of physical memory allocated to the network, reserving memory would become important.

Protocols which provide connection level flow control do this based on the amount of space in the associated socket queues. That is, send windows are calculated based on the amount of free space in the socket's receive queue, while receive windows are adjusted based on the amount of data awaiting transmission in the send queue. Care has been taken to avoid the "silly window syndrome" described in [Clark82] at both the sending and receiving ends.

12.3. Queue limiting

Incoming packets from the network are always received unless memory allocation fails. However, each Level 1 protocol input queue has an upper bound on the queue's length, and any packets exceeding that bound are discarded. It is possible for a host to be overwhelmed by excessive network traffic (for instance a host acting as a gateway from a high bandwidth network to a low bandwidth network). As a "defensive" mechanism the queue limits may be

adjusted to throttle network traffic load on a host. Consider a host willing to devote some percentage of its machine to handling network traffic. If the cost of handling an incoming packet can be calculated so that an acceptable "packet handling rate" can be determined, then input queue lengths may be dynamically adjusted based on a host's network load and the number of packets awaiting processing. Obviously, discarding packets is not a satisfactory solution to a problem such as this (simply dropping packets is likely to increase the load on a network); the queue lengths were incorporated mainly as a safeguard mechanism.

12.4. Packet forwarding

When packets can not be forwarded because of memory limitations, the system generates a "source quench" message. In addition, any other problems encountered during packet forwarding are also reflected back to the sender in the form of ICMP packets. This helps hosts avoid unneeded retransmissions.

Broadcast packets are never forwarded due to possible dire consequences. In an early stage of network development, broadcast packets were forwarded and a "routing loop" resulted in network saturation and every host on the network crashing.

13. Out of band data

Out of band data is a facility peculiar to the stream socket abstraction defined. Little agreement appears to exist as to what its semantics should be. TCP defines the notion of "urgent data" as in-line, while the NBS protocols [Burruss81] and numerous others provide a fully independent logical transmission channel along which out of band data is to be sent. In addition, the amount of the data which may be sent as an out of band message varies from protocol to protocol; everything from 1 bit to 16 bytes or more.

A stream socket's notion of out of band data has been defined as the lowest reasonable common denominator (at least reasonable in our minds); clearly this is subject to debate. Out of band data is expected to be transmitted out of the normal sequencing and flow control constraints of the data stream. A minimum of 1 byte of out of band data and one outstanding out of band message are expected to be supported by the protocol supporting a stream socket. It is a protocols perogative to support larger sized messages, or more than one outstanding out of band message at a time.

Out of band data is maintained by the protocol and usually not stored in the socket's send queue. The PRU_SENDOOB and PRU_RCVOOB requests to the *pr_usrreq* routine are used in sending and receiving data.

14. Trailer protocols

Core to core copies can be expensive. Consequently, a great deal of effort was spent in minimizing such operations. The VAX architecture provides virtual memory hardware organized in page units. To cut down on copy operations, data is kept in page sized units on page-aligned boundaries whenever possible. This allows data to be moved in memory simply by remapping the page instead of copying. The mbuf and network interface routines perform page table manipulations where needed, hiding the complexities of the VAX virtual memory hardware from higher level code.

Data enters the system in two ways: from the user, or from the network (hardware interface). When data is copied from the user's address space into the system it is deposited in pages (if sufficient data is present to fill an entire page). This encourages the user to transmit information in messages which are a multiple of the system page size.

Unfortunately, performing a similar operation when taking data from the network is very difficult. Consider the format of an incoming packet. A packet usually contains a local network header followed by one or more headers used by the high level protocols. Finally, the data, if any, follows these headers. Since the header information may be variable length, DMA'ing the eventual data for the user into a page aligned area of memory is impossible without a priori knowledge of the format (e.g. supporting only a single protocol header format).

To allow variable length header information to be present and still ensure page alignment of data, a special local network encapsulation may be used. This encapsulation, termed a *trailer protocol*, places the variable length header information after the data. A fixed size local network header is then prepended to the resultant packet. The local network header contains the size of the data portion, and a new *trailer protocol header*, inserted before the variable length information, contains the size of the variable length header information. The following trailer protocol header is used to store information regarding the variable length protocol header:

```
struct {
        short       protocol;       /* original protocol no. */
        short       length;         /* length of trailer */
};
```

The processing of the trailer protocol is very simple. On output, the local network header indicates a trailer encapsulation is being used. The protocol identifier also includes an indication of the number of data pages present (before the trailer protocol header). The trailer protocol header is initialized to contain the actual protocol and variable length header size, and appended to the data along with the variable length header information.

On input, the interface routines identify the trailer encapsulation by the protocol type stored in the local network header, then calculate the number of pages of data to find the beginning of the trailer. The trailing information is copied into a separate mbuf and linked to the front of the resultant packet.

Clearly, trailer protocols require cooperation between source and destination. In addition, they are normally cost effective only when sizable packets are used. The current scheme works because the local network encapsulation header is a fixed size, allowing DMA operations to be performed at a known offset from the first data page being received. Should the local network header be variable length this scheme fails.

Statistics collected indicate as much as 200Kb/s can be gained by using a trailer protocol with 1Kbyte packets. The average size of the variable length header was 40 bytes (the size of a minimal TCP/IP packet header). If hardware supports larger sized packets, even greater gains may be realized.

Acknowledgements

The internal structure of the system is patterned after the Xerox PUP architecture [Boggs79], while in certain places the Internet protocol family has had a great deal of influence in the design. The use of software interrupts for process invocation is based on similar facilities found in the VMS operating system. Many of the ideas related to protocol modularity, memory management, and network interfaces are based on Rob Gurwitz's TCP/IP implementation for the 4.1BSD version of UNIX on the VAX [Gurwitz81]. Greg Chesson explained his use of trailer encapsulations in Datakit, instigating their use in our system.

References

[Boggs79] Boggs, D. R., J. F. Shoch, E. A. Taft, and R. M. Metcalfe; *PUP: An Internetwork Architecture.* Report CSL-79-10. XEROX Palo Alto Research Center, July 1979.

[BBN78] Bolt Beranek and Newman; *Specification for the Interconnection of Host and IMP.* BBN Technical Report 1822. May 1978.

[Cerf78] Cerf, V. G.; The Catenet Model for Internetworking. Internet Working Group, IEN 48. July 1978.

[Clark82] Clark, D. D.; Window and Acknowledgement Strategy in TCP. Internet Working Group, IEN Draft Clark-2. March 1982.

[DEC80] Digital Equipment Corporation; *DECnet DIGITAL Network Architecture — General Description.* Order No. AA-K179A-TK. October 1980.

[Gurwitz81] Gurwitz, R. F.; VAX-UNIX Networking Support Project — Implementation Description. Internetwork Working Group, IEN 168. January 1981.

[ISO81] International Organization for Standardization. *ISO Open Systems Interconnection — Basic Reference Model.* ISO/TC 97/SC 16 N 719. August 1981.

[Joy82a] Joy, W.; Cooper, E.; Fabry, R.; Leffler, S.; and McKusick, M.; *4.2BSD System Manual.* Computer Systems Research Group, Technical Report 5. University of California, Berkeley. Draft of September 1, 1982.

[Postel79] Postel, J., ed. *DOD Standard User Datagram Protocol.* Internet Working Group, IEN 88. May 1979.

[Postel80a] Postel, J., ed. *DOD Standard Internet Protocol.* Internet Working Group, IEN 128. January 1980.

[Postel80b] Postel, J., ed. *DOD Standard Transmission Control Protocol.* Internet Working Group, IEN 129. January 1980.

[Xerox81] Xerox Corporation. *Internet Transport Protocols.* Xerox System Integration Standard 028112. December 1981.

[Zimmermann80] Zimmermann, H. OSI Reference Model — The ISO Model of Architecture for Open Systems Interconnection. IEEE Transactions on Communications. Com-28(4); 425-432. April 1980.